RADAR: Adaptive Rate Allocation in Distributed Stream Processing Systems under Bursty Workloads

Ioannis Boutsis, Vana Kalogeraki
Department of Informatics
Athens University of Economics and Business, Athens, Greece
{mpoutsis, vana}@aueb.gr

***Abstract*—In the recent years we have witnessed a proliferation of distributed stream processing systems that need to operate under bursty workloads. Examples include road traffic control, processing of financial feeds, network monitoring and real-time sensor data analysis systems. Meeting the QoS requirements of the stream processing systems under burstiness is a challenging process. In this paper we present our approach for adaptive rate allocation within the distributed stream processing system to meet the end-to-end execution time and rate demands of the applications. Our algorithm determines the rates of the application components at runtime, with respect to the QoS constraints, to compensate for delays experienced by the components or to react to sudden bursts of load. Our technique is distributed and low-cost. Our detailed experimental results over our Synergy middleware illustrate that our approach is practical, depicts good performance and has low resource overhead.**

I. Introduction

Over the recent years a number of applications that generate real-time data streams have emerged. Examples include environmental monitoring applications, medical alerting, industrial process control, network traffic monitoring, multimedia delivery, online processing of financial feeds, real-time sensor data analysis systems and location based systems for road traffic control. Distributed stream processing systems (DSPS) are commonly used platforms, comprising collections of physically distributed nodes that are connected together to offer scalable data transmission and processing functionality. In distributed stream processing systems, data produced by heterogeneous, autonomous and large numbers of globally-distributed data sources is composed dynamically and processed to generate results of interest.

Today's distributed stream processing systems are vulnerable to great variations in processing and communication delays. This is attributed to several factors: First, distributed stream processing applications need to cope with time-varying load spikes and changing demand. These are typically long running processes where the characteristics of the data may change over time and data may suddenly appear in bursts. These systems depict highly uncertain workloads which makes it difficult to predict the occurrence of a burst in advance. Worst case allocations are usually not preferred as they result in under-utilization of system resources in cases that the burst does not occur. Second, distributed stream processing systems are deployed over shared infrastructures and the resources are used concurrently by multiple executing applications. Thus, the occurrence of a burst at one application can severely delay processing and affect the performance of multiple applications in the system. The third challenge comes from the fact that in a distributed stream processing system centralized approaches are not efficient as it is difficult to maintain an accurate view of the system at all times, since the time to communicate the state of the system to a centralized manager is large and as a result they cannot accurately capture the changes in the availability of the system resources. Thus, a fundamental challenge to the effective operation of these systems is their ability to identify events of interest in *real-time*, even in the face of highly bursty loads.

While burst management has recently identified as being critical, previous techniques are not sufficient as they either result in underutilization of system resources or may require a substantial amount of packet drops upon the onset of a burst. One common approach to deal with the problem of overload is to apply admission control or static reservation techniques [1]. However, such over-provisioning is of limited use as it can have significant cost in terms of underutilized nodes. Furthermore, data bursts can happen and must be dealt with at runtime. Feedback control approaches have also been proposed. However, feedback control approaches use coarse-grained models where they adjust the overall CPU utilization on the processors in order to meet the application timeliness requirements. These are best used to control the aggregate performance of a distributed system, rather than building flexibility in the system parameters in terms of the application parameters. A more subtle approach is to determine a set of *feasible* solutions during an offline phase, using approaches such as MultiParametric Rate Adaptation [2] or using Pareto points[3], so that the system can select and trigger one of these solutions during the online phase based on application demands and resource availability. These approaches [2] [4], including work from our own group [3], have shown that there is a benefit in solving the problem online, taking into account estimates of application latencies and slack times, along with current

resource utilization measurements, to schedule the execution of the different applications, and adjust the estimates based on bursty load and resource availability. Online techniques have significant advantage over offline approaches, that, although produce optimal solutions, they are computationally expensive and need to know in advance complete system information, which is often not possible in Internet-scale systems. One common online optimization solution is to apply load shedding [5]; however, this randomly drops data units at certain time points based on resource thresholds, in order to reduce the load without considering the importance of the data and as a result significant information may get lost.

In this paper we investigate the problem of adaptive rate allocation within a DSPS in order to meet the end-to-end execution time requirements of distributed stream processing applications under burstiness. We summarize our contributions below:

- We present **RADAR**, our system that dynamically determines the data rates of the application components on the nodes of a DSPS under bursty workloads. RADAR uses measurements of elapsed times, application projected latencies and measurements of resource loads to dynamically determine the rate allocation for distributed real-time stream processing applications to meet their end-to-end timeliness and rate demands.
- We develop a distributed constrained optimization formulation of the problem based on the Lagrange Multipliers technique, and provide a distributed and low-cost method to solve it. Our approach manages to solve the optimization problem in a distributed manner, where the initial global optimization problem is decomposed into a set of subproblems, that can be solved with local processing and communication among the components of the applications. Moreover, it renders the rate allocation optimization problem amenable to completely decentralized implementation, where a global optimization problem is turned into a local one, whose solution requires only low-overhead coordination between the nodes involved.
- We have run extensive experiments over PlanetLab [6] to validate our approach. Our experimental results illustrate that our approach effectively meets application real-time demands, depicts good performance and outperforms its competitors.

II. System Architecture and Model

A. The Synergy System

We have built our approach in our distributed stream processing system called Synergy[7]. Synergy is a wide-area stream processing middleware that comprises a set of distributed nodes denoted as n_i, connected via virtual links denoted as l_j. The goal of Synergy is to support the

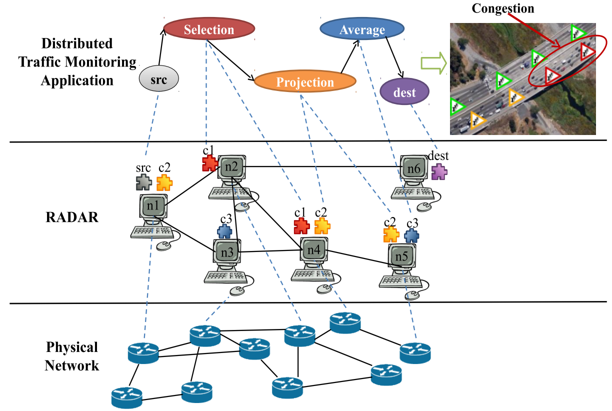

Figure 1. Our system architecture.

execution of distributed stream processing applications with QoS constraints, while efficiently managing the system's resources. Synergy is built as an overlay on top of the Pastry peer-to-peer network and runs on Planetlab [6]; it leverages the Pastry location and routing protocols for registering and discovering available application components and streams in a scalable manner. It implements a *monitoring module* for building resource utilization profiles (CPU load, network bandwidth) and maintaining application latency measurements and a fully distributed *composition module* that selects and instantiates application components at runtime. Figure 1 shows the Synergy architecture.

B. System Model

A distributed stream processing application in the system is modeled as an application graph, where each node in the graph represents a service invoked by the application. Each service represents a processing function. The instantiation of a service on a node is called a *component*; a service can be instantiated at multiple nodes in the system and invoked by different applications. A component operates on individual chunks of data, named *Application Data Units*(ADU). The size of a data unit depends on the type of application. Examples of data units are sets of measured values (such as $<$ timestamp, Latitude, Longitude, speed $>$ that we used in our traffic monitoring application) or can be sequences of picture or audio frames (for example, in a multimedia application). A distributed stream processing application is executed collaboratively by instantiating the appropriate components on the Synergy nodes.

The user submits a request for a *set of q applications*, $1 \leq q \leq Q$ to one of the nodes in the system, along with their respective *initial rate requirements* r_q. The initial rate requirement r_q for an application represents the delivery rate of data units requested by the application. The application is characterized by its deadline $Deadline_q$ that represents the time interval within which the application q should complete the end-to-end processing for the ADUs of the rate determined. When submitting a request, the user expects from the system to instantiate the appropriate components

on the system in order to perform the processing required by each application, at the rate requested by the application.

Upon reception of a data unit by a node, the data unit is inserted in the scheduler's queue waiting to be processed. The order by which data units are processed, is defined by the scheduling policy (we currently use FIFO).

Each invoked component c_i is characterized by its resource requirements $u_{c_i}(j)$ 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.

III. The Problem

A. General Optimization Problem

Our general problem formulation develops a constrained optimization problem. Let C be the set of components that can be deployed in the system, and R_{c_i} be the rate that each component c_i will get. Component rates are constrained by the rate of the application that invokes them, but their actual rates depend on the selectivity of the components. Every application q is represented as a graph of component invocations that executes on an input data stream and needs to finish within $Deadline_q$, which is a relative value. Our main objective is to maximize the rates for the components invoked by the applications in the system so that the deadline requirements of the applications are met and the resource constraints are satisfied. This is described as:

$$maximize \sum_{c_i \in C} R_{c_i} \tag{1}$$

Our maximization function suggests that we choose to maximize the rates for the components so as to maximize the system utilization, by identifying the rates that can be supported by all components in each application graph. The summation of the components' rates will be maximized with respect to the following constraints:

End-to-End Target Time Constraint. Our goal is to meet the end-to-end deadlines of the applications. In order to ensure that the application executes within its $Deadline_q$, the sum of the computation times of all the components invoked by the application q and the corresponding communication times (that we denote end-to-end execution time) need to be smaller that the application deadline. This can be expressed as follows:

$$ExecTime_q \leq Deadline_q \tag{2}$$

where:

$$ExecTime_q = max_{path}(\sum_{c_i \in q} Comp_{c_i}(R_{c_i}) * CPU_{c_i} + \sum_{c_i \in q} Comm_{c_i -> c_{i+1}}(R_{c_i})) \tag{3}$$

where max_{path} is used in the case that the application is represented as a graph with more than one paths, so that the end-to-end execution time of the application is the maximum path latency. In the above equation, $Comp_i(R_{c_i})$ represents the average computation time required for component c_i to execute at rate R_{c_i}, obtained through profiling techniques with low overhead, and $Comm_{c_i -> c_{i+1}}(R_{c_i})$ represents the corresponding communication time between components c_i and c_{i+1}. CPU_{c_i} denotes the average percentage of CPU for component c_i to process one ADU and depends on the scheduling policy used. In our technique we use FIFO scheduling policy.

Resource Constraint. All components on a processor are competing for available CPU resources. This constraint states that the sum of the rates allocated to each component multiplied by the CPU share required to process each ADU must be smaller than the fraction of available resources. For each node we denote:

$$\sum_{c_i \in n} R_{c_i} * CPU_{c_i} \leq 1 \quad \forall n \in Nodes \tag{4}$$

We focus our attention to the processing resource (CPU capacity), as this is the sparsest resource in processing intensive environments such as the distributed stream processing systems that we consider. CPU_{c_i} represents the average percentage of CPU, for component c_i to process one ADU, within $Deadline_q$.

Flow Conservation Constraint. 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. Then the flow conservation constraints are represented as:

$$R_{c_i+1} = sel_{c_i} * R_{c_i} \quad \forall c_i \in Components \tag{5}$$

Rate Constraint. Moreover, we denote minimum and maximum rate constraint for each component that should be defined before instantiating the component :

$$MinRate_{c_i} \leq R_{c_i} \leq MaxRate_{c_i} \quad \forall c_i \in Components \tag{6}$$

Discussion. It is also important to notice that the burst occurring at one application component of a node can affect multiple applications running components on that node. This can potentially create queueing delays at components running in remote nodes. Thus, RADAR solves the optimization problem not only for the bursty node, but for all nodes

affected by the burstiness, that is, all nodes running components of the affected applications. When a burst occurs, RADAR uses the maximization function (1) to define the rates of the components on all affected nodes. The resource, rate and flow conservation constraints are given by equations (4), (5) and (6) respectively. For each application executing components on the *affected* nodes RADAR computes the end-to-end execution time of the applications using equation (2) to determine whether the latencies of the applications are too high causing the applications to miss their deadlines. In the next section we describe our distributed technique for solving this constrained optimization problem.

B. Distributed Optimization Formulation

Observation 1. *In our framework we can consider several functions to model the computation time of the components as a function of the data rate. However, works like [8] have shown that there is a linear relationship of the execution time and the data rate, in other words the running time can be expressed as a linear function of the input data rate of the component. This can be expressed with the following equation:*

$$Comp_{c_i}(R_{c_i}) = C_{c_i} * R_{c_i} \tag{7}$$

where C_{c_i} represents the computation time of component c_i per data unit. Similarly, we can consider communication time $Comm_{c_i->c_{i+1}}(R_{c_i})$ as a linear function of the component's input data rate.

Lemma 1. *Given Observation 1, our optimization problem is an instance of linear optimization (LP).*

Our optimization problem is optimizing equation (1) given constraints (2), (4), (5), (6). We note that equation(2) can be expressed using (3) and (7), and for equation (3) the maximum can be computed once, before forming the optimization problem because it is based on the graph that remains static during our optimization step.

Solving the problem with a Distributed Technique: To solve the constrained maximization problem we first convert it to an unconstrained maximization problem by the method of Lagrange multipliers [9], [10]. This method allows us to relax the constraints, i.e. add the equality and the inequality constraints to the objective function by multiplying them by some non-negative values, the Lagrange multipliers and the KKT multipliers respectively, which can be viewed as penalties for violating the constraints. There must be one Lagrange multiplier for each equality constraint and one KKT multiplier for each inequality constraint.

We define that the system is congested when at least one of the following constraints is violated (*i.e.,* $ExecTime_q > Deadline_q$ or $\sum_{c_i \in n} R_{c_i} * CPU_{c_i} > 1$). In the dynamic environment that we consider, application rates may increase or decrease dynamically without a priori notification. This can lead to congestion or underutilization of the system resources respectively. Thus, the system must be able to adapt to such dynamic workloads quickly, by implementing optimization solutions that are highly efficient at runtime, to recompute the highest rates that the system can provide to each component based on the global constraints. This is done in a distributed and iterative manner using the Lagrange Multipliers method.

Lagrange Multipliers Method. We define the Lagrangian of the original optimization problem as:

$$\begin{aligned} L = \sum_{c_i \in C} R_{c_i} - \sum_{q \in Apps} \lambda_q * (ExecTime_q - Deadline_q) \\ - \sum_{n \in Nodes} \lambda_n * (\sum_{c_i \in n} R_{c_i} * CPU_{c_i} - 1) \\ - \sum_{c_i \in C} \gamma_{c_i} * (R_{c_i+1} - sel_{c_i} * R_{c_i}) \end{aligned} \tag{8}$$

where λ_q, λ_n and γ_{c_i} are the Lagrange and KKT multipliers. We notice that both the objective and the constraint functions, expressed in terms of component rates, are concave and continuously differentiable, in the region where the constraints are satisfied. Hence, finding the maximum for the objective function is equivalent to finding the maximum for the Lagrangian, subject to $\lambda_q, \lambda_n \geq 0$, due to the strong duality. Thus, instead of solving the original optimization problem, we solve the alternative problem for each component c_i, given specific values for λ_q, λ_n and γ_{c_i}.

In our case, since L is differentiable, the problem can be solved using the gradient method, that works by iteratively updating the values of L's variables until L converges to some value. Moreover, computing the λ_q, λ_n, γ_{c_i} is also done in a distributed manner, since every variable can be computed in a different node. We note that the constraint from equation (6) is imposed independently during iterations (similar to [11]). All of the components compute their new rate R_{c_i} independently at every iteration. After computing its rate every component should set its value as $min(max(R_{c_i}, minRate_{c_i}), maxRate_{c_i})$.

Note though, that, the nodes that participate in the above process are only the ones that host components of the application graph that experience the burst. If multiple application graphs experience bursts, we run the optimization algorithm for each graph separately.

IV. Experimental Evaluation

A. Experimental Setup

We have implemented RADAR over the Synergy middleware [7] and tested it on PlanetLab [6]. RADAR is written in Java6u26 with approximately 12K lines of code. The experimental evaluation focuses on the following parameters: (i) **Application Rates**, (ii) **Application End-to-end Execution Times**, (iii) **Number of Deadline Misses**, (iv) **Throughput**, and (v) **Scalability**. We compare our approach with the **Load Shedding** technique[5] which is a commonly used online technique to address bursts. Load shedding aims to reduce load by dropping data units at certain time points based on CPU load thresholds. For the experiments, we implemented a traffic monitoring application using the Berkeley's Mobile Millennium Dataset [12].

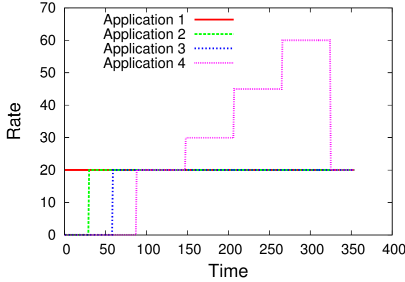

Figure 2. Application injected rates.

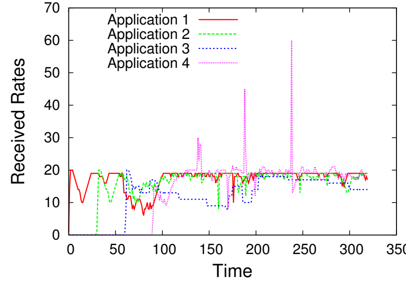

Figure 3. Rates using RADAR.

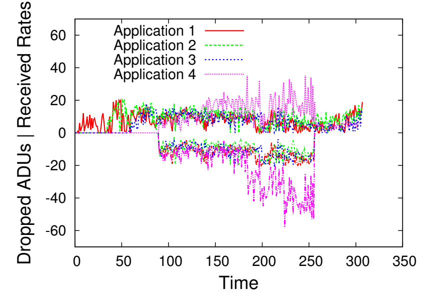

Figure 4. Rates using L.S.

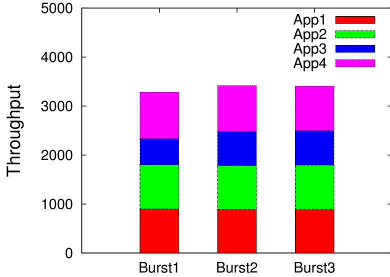

Figure 5. RADAR Throughput over Different Burst Intensities.

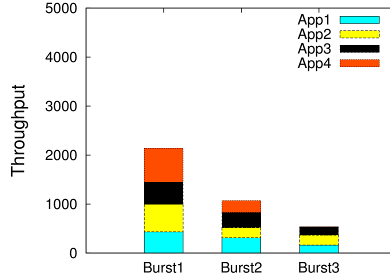

Figure 6. Load Shedding Throughput over Different Burst Intensities.

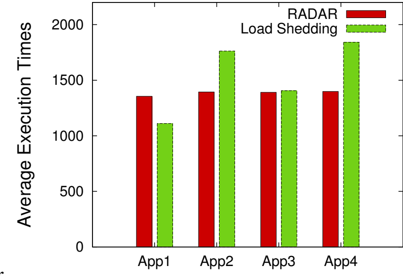

Figure 7. Average end-to-end execution times.

B. Experimental Results

In this section we evaluate the operation of RADAR under burstiness. We used 4 applications and a total of 16 application components. The applications start 30 seconds apart so that we are able to observe the behavior of the other applications when a new one starts. The deadlines of the applications are set to 1400ms. All applications request a minimum rate of 5 and a maximum of 20, while the fourth application is the bursty one where we inject the additional workload so its maximum rate has to be modified on runtime. We simulate three different burst intensities, where the rate of application 4 suddenly increases to 30, 45 and 60 ADUs per deadline time. Figure 2 shows the injected rates of the applications during the experiment's lifetime.

Application Rates. Figure 3 shows the rates received by the applications using RADAR. All applications start with a maximum rate of 20 and then they adjust their rates using our approach, based on the applications running on the system and the availability of system resources. As the applications execute, they are scheduled according to their requested rates. RADAR monitors the application execution and computes their end-to-end execution times. As one can observe, when application 4 starts experiencing bursts, RADAR triggers the distributed optimization algorithm and adjusts the rates of application 4 to its maximum possible among requested rates. Thus, our approach manages to keep the application rates within their demands at all times.

Figure 4 shows the rates received by the applications using the Load Shedding technique. What is shown in the negative axis in the figure is the corresponding number of data unit drops caused due to Load Shedding for the different applications during the bursts. Load Shedding deals with the problem of the bursty workload by dropping ADUs. As a result, a large number of ADUs are dropped when bursts occur. Load shedding violates the minimum rate requirements since 24% of the total rates were below the rate requirement of the applications. This happens because overload might occur in multiple nodes in the system and several applications may get affected. Load shedding reacts to the overload by dropping ADUs from all applications in the affected nodes, that will result in a large number of ADUs being dropped in parallel on multiple nodes, which might be more than needed to deal with the overload. Furthermore, the dropped ADUs will have already used resources on previous nodes, further contributing to the problem of overload.

Throughput. Figures 5 and 6 illustrate the total throughput achieved in the system during the burst intervals, along with a breakdown of the throughput of each application, for both techniques respectively. Our algorithm adjusts the rates to react to the bursts. As a result the bursts have a small impact to the system throughput and the impact is constrained because RADAR triggers and eliminates overload upon the onset of the burst. On the contrary, the throughput is much smaller using the Load Shedding technique because when the burst occurs, the CPU Load on the node increases and as a result Load Shedding drops a large number of ADUs to sustain the requested CPU thresholds. As we explained before, even though the burst happens at application 4, it affects multiple applications running in the system nodes.

Figure 7 presents the average end-to-end execution times of all applications running in the system using both techniques. As the figure illustrates, RADAR achieves end-to-end execution times that are much closer to the Deadline compared to the Load Shedding technique, which may complete the execution of the ADUs quicker, but at the expense of package dropping. As can be observed, using RADAR, all applications managed to have average execution times below the Deadline with an average of 1383ms. On the contrary, Load Shedding had higher execution times with a total average of 1616ms.

V. Related Work

Distributed stream processing systems have recently become extremely popular for processing high-throughput, low-latency data streams and a number of such systems have emerged in the literature (including our own work [7], [13]).

Optimal service composition is proposed in [13] where it uses a probing protocol and coarse grained global knowledge to achieve optimal load balancing among the nodes. Our work targets to maximize the rates, while keeping the QoS within the users requirements. Amini *et al* in [14] assume a DSPS similar to Synergy. Their method relies on feedback from downstream components and the goal is to maximize a global system utility. Authors in [15] propose a bin-packing solution in order to solve the resource allocation problem that implies a centralized solution. Wen *et al* in [16] propose a distributed technique using Lagrange multipliers like ours, to maximize the P2P streaming in Wireless Mesh Networks. Nonetheless, their method considers only the link rate allocation problem among peers.

Lumezanu *et al* in [4], model the latency assignment problem for real-time distributed applications as a utility maximization problem. However, their solution has high overhead as it runs constantly to adjust the parameters in the system and for all nodes, as opposed to RADAR that runs only when a burst is detected using as few nodes as possible. In [17], they apply a well-known network routing algorithm to allocate resources on stream processing nodes. Their method achieves optimal allocation of computing and bandwidth resources, however it needs time to be effective.

Finally, recent efforts have studied the problem of overloads in DSPS. In our previous work [3], we have proposed the BARRE algorithm for accommodating unpredictable bursts of the data streams in DSPS. BARRE relies on the use of an offline phase, where complete system information is needed in advance. In a similar approach, authors in [2] also use an offline computation phase for the optimization. Authors in [5] consider the problem of how to avoid overloads in distributed stream processing systems though load shedding. However, our experimental results illustrate that RADAR outperforms load shedding techniques. In [18] they also deal with bursts using a nonprobabilistic model. They suggest that data streams that enter the system should satisfy the burstiness constraint to reduce network delays.

VI. Conclusions

In this paper, we have presented RADAR, our system for adaptive rate allocation within the Distributed Stream Processing System in order to meet the end-to-end execution time and rate demands of the applications. Our approach determines the rates for each application's components at runtime, with respect to the application real-time demands, to compensate for queueing delays experienced by the components or to react to sudden bursts of load, in a distributed manner. Our experimental results illustrate that RADAR is practical, scalable, depicts good performance and outperforms its competitors.

ACKNOWLEDGEMENT: This research has been supported by the European Union and Hellenic Republic Ministry of Education, Lifelong Learning and Religious Affairs through the Thalis DISFER project, through the Marie-Curie RTD (IRG-231038) project and by AUEB through a PEVE project.

References

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

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

[3] Y. Drougas and V. Kalogeraki, "Accommodating bursts in distributed stream processing systems," in *IPDPS*, Rome, Italy, May 2009, pp. 1 – 10.

[4] C. Lumezanu, S. Bhola, and M. Astley, "Online optimization for latency assignment in distributed real-time systems," in *ICDCS*, Beijing, China, June 2008, pp. 752–759.

[5] N. Tatbul, U. Çetintemel, and S. Zdonik, "Staying FIT: Efficient load shedding techniques for distributed stream processing," in *Proc. of VLDB*, Vienna, Austria, Sep. 2007.

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

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

[8] Y. Wei, V. Prasad, S. Son, and J. Stankovic, "Prediction-based QoS management for real-time data streams," in *Proceedings of 27th RTSS*, Rio de Janeiro, Brazil, December 2006.

[9] D. P. Bertsekas, *Nonlinear Programming*. Belmont, MA: Athena Scientific, 1999.

[10] D. P. Palomar and M. Chiang, "A tutorial on decomposition methods for network utility maximization," *IEEE Selected Areas in Communications*, vol. 24, pp. 1439–1451, 2006.

[11] S. H. Low and D. E. Lapsley, "Optimization flow control, i: Basic algorithm and convergence," *IEEE/ACM Transactions on Networking*, vol. 7, no. 6, pp. 861–874, 1999.

[12] J. Herrera, "Evaluation of traffic data obtained via gps-enabled mobile phones: The mobile century field experiment," in *Transport. Res. Part C*, 2009.

[13] X. Gu, P. S. Yu, and K. Nahrstedt, "Optimal component composition for scalable stream processing," in *25th ICDCS*, Columbus, OH, 2005.

[14] L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure, "Adaptive control of extreme-scale stream processing systems," in *Proc. of 26th ICDCS*, Lisboa, Portugal, July 2006.

[15] N. Roy, J. S. Kinnebrew, N. Shankaran, G. Biswas, and D. C. Schmidt, "Toward effective multi-capacity resource allocation in distributed real-time and embedded systems," in *Proceedings of 11th ISORC*, Orlando, Florida, May 2008.

[16] J. Wen, J. Cao, K. Xie, and R. Li, "User density sensitive p2p streaming in wireless mesh networks," *Parallel and Distributed Computing*, vol. 71, pp. 573–583, 2011.

[17] C. H. Xia, D. Towsley, and C. Zhang, "Distributed resource management and admission control of stream processing systems with max utility," in *Proceedings of the 27th ICDCS*, Toronto, Canada, June 2007, p. 68.

[18] R. Cruz, "A calculus for network delay. i. network elements in isolation & ii. network analysis," *IEEE Transactions on Information Theory*, vol. 37, no. 1, pp. 114 – 141, Jan 1991.