Resource Management using Pattern-based Prediction to Address Bursty Data Streams

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 efficiently, even when data bursts occur. Examples include road traffic networks, processing of financial feeds, network monitoring and real-time sensor data analysis systems. An important challenge in managing these systems is effective resource management and meeting the QoS demands of the stream processing applications under different workload conditions, even under bursts. In this paper we present our approach that aims to predict the execution times of the distributed stream processing applications by taking into account the effects of the bursts and what is the typical workload of the stream processing system. Our approach builds application data rate patterns at run-time and predicts the effect of the burst on the performance of the applications, to identify whether there is a need to react on the onset of a burst. 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, we have observed a rapid growth of applications that need to operate on continuous real-time data streams, where the stream data are typically generated from sensors embedded on ubiquitous devices, such as smartphones, tablets, etc. Examples of such applications include transportation systems, medical alerting, network traffic monitoring systems, multimedia streaming, online processing of financial feeds, real-time sensor data analysis systems and environmental monitoring systems. These are typically fed into a Distributed Stream Processing System (DSPS) [1] and processed in real-time in order to identify events of interest.

Various factors make the management of these systems challenging in practice. First, the majority of the applications in the distributed stream processing systems have inherent time constraints on the execution of the data (*i.e.,* detecting congestion on a road network). However, meeting the QoS constraints, *i.e.,* real-time constraints, of the applications is a challenging task, since (1) the workload in these systems can vary due to the dynamic nature of the data streams, and as a result the applications need to cope with time-varying load spikes and often bursty workloads, (2) the occurrence of a burst is difficult to be predicted in advance, and (3) the applications are deployed over shared infrastructures and thus they compete among each other for system resources when executing concurrently. Furthermore, these are complex decisions, where the actions taken to address changes on one application may significantly affect the remaining applications, and thus require significant planning. Hence, a fundamental challenge to the effective operation of the DSPSs is their ability to manage the applications and the resources efficiently, even in the face of unpredictable events such as load spikes or highly bursty data.

Burst management has recently been identified as being critical and several techniques have been proposed. One common online solution is to apply load shedding [2]; however, this randomly drops data units 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. Feedback control approaches (such as the work in [3]) are also not efficient as 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. In our previous work we have proposed a distributed solution that uses measurements of elapsed times, application projected latencies and measurements of resource loads to dynamically determine a new rate allocation adjustment for distributed real-time stream processing applications, to react to bursty situations [4]. We have also proposed BARRE [5] that relies on an offline phase to identify all feasible allocations that we can choose at runtime. All the above approaches are either reactive or use offline information, and they do not consider the effect for reacting to the burst, especially when bursts are transient and may occur only for short periods of times.

In this paper we present an approach that aims to estimate the impact of the bursts on the response times of the distributed stream processing applications, when multiple applications concurrently execute on shared system resources. Our approach identifies application data rate patterns at runtime, builds profiles of the execution times of the applications, and estimates what is the effect of bursts on the performance of the applications which are co-located on the same nodes with the bursty ones. Hence, calculating the expected load, enables us to predict whether there is a need to react upon the onset of a burst so as to meet the QoS constraints of the applications. We envision this approach as a valuable tool for the efficient management of the system, that executes complementary to the exhaustive burst management techniques and triggers them only if necessary, since it has a lower processing cost and is more effective in cases where a feasible load will be achieved shortly after the bursts occur.

We summarize our contributions below:

- We present a prediction-based approach, that identifies application data patterns dynamically, based on the historical applications executions. It uses measurements of the computational and communication elapsed times and predicts whether there is a need to react on bursty situations so that the applications meet their end-to-end timeliness and rate demands.

- We provide a formulation to estimate the execution times of the applications in the distributed stream processing system, based on the rate patterns of the applications.
- We integrate our proposed approach in our distributed stream processing system Synergy [1]. 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, is practical and depicts good performance.

II. System Architecture and Model

A. The Synergy System

In this section we give a brief description of Synergy [1], our distributed stream processing system, where we have implemented our approach. Synergy is a wide-area stream processing middleware. It comprises a set of distributed nodes, connected via virtual links, so that the execution of distributed stream processing applications is supported, and the systems resources are efficiently managed. Synergy is built over the Pastry network [7], as an overlay, and runs on Planetlab [6]; it utilizes the Pastry location and routing protocols to register and discover the available application components and streams in a scalable manner.

Each Synergy node consists of the following main modules: (i) A *discovery module* that identifies the available application components and data streams in the system. Synergy leverages the capabilities of the Pastry overlay network for registering and discovering available components and streams in a decentralized manner. In our current prototype we have implemented a keyword-based discovery service, on top of the Pastry distributed hash table (DHT). This allows us to register and discover components by hashing keywords instead of the component IDs themselves, and thus decouple component placement from their discovery. ii) A *routing module* is responsible to route data streams and protocol messages between nodes. iii) A *monitoring module* that builds resource utilization profiles (cpu load, network bandwidth) and maintains application latency measurements. iv) A *composition module* that implements a fully distributed composition protocol that selects and instantiates application components. v) An *application module* that implements the application logic. We have implemented two applications in our system: real-time traffic using the Berkeley Mobile Millennium dataset [8] and stream encryption. vi) A *replica placement module* that determines the replication and placement of component replicas on the Synergy nodes in order to maximize application availability. vii) A *scheduling module* that implements various scheduling algorithms on the Synergy nodes (we currently implement FIFO, Earliest Deadline First and Low Level Scheduling). viii) A *burst management module*, that attempts to dynamically determine the rate allocation to react to bursts. ix) We extend Synergy with a *Prediction module* that estimates application execution times by considering the application data patterns. Figure 1 shows the Synergy architecture.

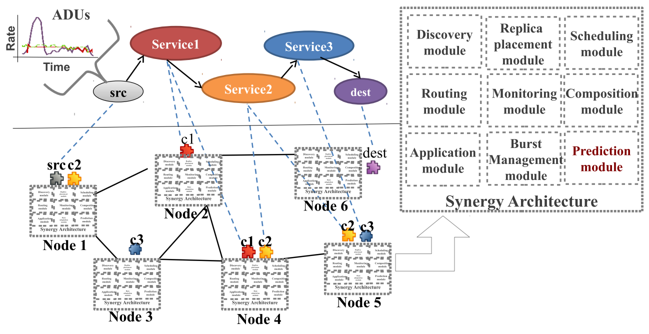

Fig. 1. Our system architecture.

B. System Model

In Synergy, we model the distributed stream processing applications as application graphs where the nodes in a graph represent the services (processing functions), invoked by the application. The instantiation of a service on a node is called a *component*; each service can be instantiated at several nodes in the system and can be invoked by multiple applications. Every component operates on individual chunks of stream data, called *Application Data Units* (ADU), where the size of each data unit depends on the type of the application. The data units are composed from a set of measured values, such as $<$ timestamp, latitude, longitude, speed $>$ that we used in our traffic monitoring application or $<$ timestamp, latitude, longitude, accelerometer values, microphone samples $>$ that can be used for earthquake monitoring. A distributed stream processing application is executed collaboratively by instantiating the appropriate components on the nodes of Synergy. Every node that participates in the Synergy DSPS is responsible to host several services.

The user submits a request for a *set of q applications*, $1 \leq q \leq Q$ to one of the nodes in the system. Each application q is characterized by:

- The *initial rate requirements* R_q that represents the input rate of the application data units. When submitting a request, the user expects from Synergy to instantiate the appropriate components to perform the required processing for the applications, at their requested rates.
- A $Deadline_q$, a relative value, that represents the time interval within which application q should complete the execution of the data units end-to-end, at the requested rate.

Upon reception of a data unit by a node, the data unit is inserted at the queue of the scheduler, waiting to be processed. The order by which the data units are going to be processed, depends on the selected scheduling policy (in our experiments we used FIFO).

The input rate R_q of each application q over time, is stored by the Prediction module. Thus, R_q is used to build profiles of the application rates. The application rate profiling is based on the observation that the application rates tend to repeat over

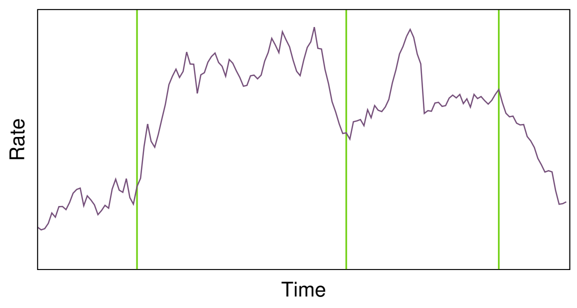

Fig. 2. Example Pattern.

time periods. Thus, we take advantage of this repetition by identifying the most frequent patterns for each time period. Hence, the rate R_q of the application q is calculated through these patterns as will be discussed in the following section. An example of a pattern is illustrated in figure 2, obtained from a traffic monitoring application in our stream processing system (we discuss the traffic monitoring application we used in the experimental evaluation section in detail).

Let C be the set of the deployed components in Synergy, and the rate that each component $c \in C$ receives, is denoted as R_c. Every component's rate is constrained by the input rate of the application R_q that invokes the component, since the actual rates depend on the selectivity of the components. Thus, we can use the input rate R_q of the application q, that is extracted from the pattern, to determine the rate R_c, for each component in application q.

Each invoked component c in the Synergy stream processing system is characterized by:

- Its resource requirements $U_c(j)$ for each resource j it uses, such as CPU or bandwidth.
- Its selectivity sel_c that represents the ratio of the output to the input rate of the component.

Both the rate requirements and the selectivity of a component, are characteristics of the service executed by the component. These can be either provided by the user prior to the application execution or can be acquired at run-time through profiling [1]. Also, note that, the execution of a service can be assigned to more than one components for a specific application. In that case each component will be responsible for a subset of the data that will be processed.

III. Our Proposed Approach

A. Identifying the Pattern

In many of the stream processing applications we observe that the input rates follow specific patterns [9], [10], [11] over time periods. Let R be a set of records for a specific application, where each record consists of a timestamp and the respective input rate of the application. Let $I = i_1, i_2, ..., i_m$ be the set of every available input rate, defined as Rates. A rate-set is a non-empty set of Rates and a sequence s is a set of rate-sets ordered according to their timestamp, as $< s_1\ s_2\ ...\ s_n >$, where $s_j, j \in 1...n$, is a rate-set. For instance, if an application achieved rates "45, 40, 38", the respective sequence can be described as: $s_1 =< (45), (40), (38) >$.

A sequence $S' =< s'_1\ s'_2\ ...\ s'_n >$ is a subsequence of another sequence $S =< s_1\ s_2\ ...\ s_m >$, denoted $S' \succ S$, if there exist integers $i_1 < i_2 < ...i_j... < i_n$ such that $s'_1 \subseteq si_1$, $s'_2 \subseteq si_2$, ... $s'_n \subseteq si_n$. For example the sequence $s_2 =< (45), (38) >$ is a subsequence of s_1, described as $s_2 \succ s_1$ since $(45) \subseteq (45)$ and $(38) \subseteq (38)$.

All the application rates, from the same application, for a specific time window, are grouped together, sorted in an increasing order and denoted as a data sequence. A data sequence contains a sequence S if S is a subsequence of the data sequence. Our goal is to define the data sequences which are repeated frequently. Hence, we need to identify the sequences that contain more than a predefined amount of subsequences, that is denoted as a *sequential pattern*, also called *frequent sequence*. A few solutions exist in the literature for the problem of sequential pattern mining [9], [10], [11].

We follow the technique proposed in [9] to identify the sequential patterns, because it uses a novel data structure to incrementally maintain frequent sequential patterns and a fast pruning strategy. Moreover, it enables users to issue requests for frequent sequences over an arbitrary time interval, at any time. This approach uses a tree structure for inserting the sequences and the key idea is to maintain only the maximal subsequences for each time window into the tree. Thus, the search space is reduced when comparing and pruning sequences by maintaining only a minimal number of sequences needed to answer queries. Every new sequence is either inserted to an existing valuation, if it belongs or extends an existing sequence, or can be inserted as a new valuation if it is not a subsequence of the existing ones. Finally, the sequential patterns can be identified rapidly from the maximal subsequences for each time window. Nevertheless, any solution to this problem can be plugged in to our system.

Our approach takes advantage of the sequential patterns to predict the forthcoming rates of the applications. Thus, we can identify the sequential patterns for each time window that we are interested in, by analyzing the sequences that correspond to that window. For example, if we want to identify the most frequent sequences per hour of the day, we should create the sequences that contain the rates of the application for each individual hourly time period, (i.e. 2.00am-3.00am), that can be obtained from historical measurements. The window is typically determined by the system developer since it depends on the characteristics of the individual applications. For instance, we may need to identify the sequential pattern for each individual minute of the day if the workload is highly unstable, or for every hour that corresponds on a specific day of the week, as the rates may change among the days of the week. After identifying the most frequent sequence we can predict the forthcoming input rates of each application for the time window that we investigate. Thus, for each timepoint $t \in window$ we define the corresponding R_q that represents the respective rate for application q, that can be extracted from the sequential pattern, to estimate the future incoming traffic.

Computing the sequential patterns online, when a burst occurs can be computational costly, depending on the selected solution, and should be avoided when the system is congested.

Thus, we compute them during time periods where the system is underutilized, to reduce the overhead.

B. Estimating Execution Times

Once we have identified the most frequent patterns, we use them to determine if the the estimated forthcoming rates can be sustained by the system, even after the burst. Thus, we develop a formulation based on the system's resource constraints.

The relation between the input and the output rates of a component, is defined by the selectivity of the component. The selectivity sel_c of a component c represents the average ratio of the number of output data units to the number of input data units of c and depends on the service run by the component. Thus, the input rate of each component depends on the selectivity and the input rate of the previous component in the application graph which is represented as:

$$R_{c+1} = sel_c * R_c \quad \forall c \in C \tag{1}$$

Our objective is to determine whether the rates of the components invoked by the applications in Synergy satisfy the deadline requirements of the applications and the resource constraints, even after the burst. The constraints that we need to satisfy are the following:

End-to-End Time Constraint: The first constraint is developed to ensure that the applications meet their end-to-end deadlines. Every application q should execute within its $Deadline_q$, meaning that the sum of the computation times and the corresponding communication times of all components invoked by application q (denoted as end-to-end execution time) should be less than the $Deadline_q$. This can be expressed as follows:

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

where:

$$ExecTime_q = max_{path}(\sum_{c \in q} Comp_c(R_c) * (\frac{1}{P_c}) + \sum_{c \in q} Comm_{c \rightarrow c+1}(R_c) * (\frac{1}{L_c})) \tag{3}$$

where max_{path} is used in the case that the application is represented as a graph with multiple paths, so that the application's end-to-end execution time is the maximum path latency. In the above equation, $Comp_i(R_c)$ represents the mean execution time required for component c to execute at rate R_c, obtained through profiling techniques, and $Comm_{c \rightarrow c+1}(R_c)$ represents the corresponding mean communication time among components c and $c+1$. P_c denotes the CPU share on the local processor for the execution of component c, while L_c corresponds to the corresponding share on the communication link for c. The end-to-end execution time, in the type of applications that we consider, is mainly attributed to the execution times of the components and the communication times are negligible.

Several works have shown that there is a linear relationship among the execution time and the data rate[4] [12] [13], meaning that the execution time can be expressed as a linear function of the input data rate of the component. This can be expressed as: $Comp_c(R_c) = C_c * R_c$, where C_c represents the computation time of component c per data unit. Similarly, we express the communication time $Comm_{c \rightarrow c+1}(R_c)$ as a linear function of the component's input data rate.

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

$$\sum_{c \in n} R_c * CPU_c \leq 1 \quad \forall n \in Nodes \tag{4}$$

where CPU_c represents the average percentage of CPU, for component c to process one ADU, within $Deadline_q$.

C. The Prediction Algorithm

Our algorithm takes advantage of the predicted application patterns, discussed in the previous section, to investigate the need of reaction under burstiness. Thus, we need to verify whether the predicted rates of the applications that participate in the application graph of the bursty application, will satisfy the QoS constraints of the applications. Our experience has shown that the workload in the Distributed Stream Processing Systems varies. Moreover, the rates of a component affect through queueing multiple components. Thus, we aim to estimate if the burst experienced by an application is going to affect other applications that are co-located on the same nodes, keeping in mind that the rates can change over time for all the components. Hence, we use the rate patterns to determine whether the end-to-end execution times of the applications are going to meet their deadline constraints for a time window after the occurrence of a burst.

Our algorithm is triggered by the monitoring module of Synergy that identifies the burst, this typically occurs when the application missed its deadline. Thus, whenever the algorithm is triggered, the Prediction component is initiated. The Prediction component estimates the forthcoming rates based on the applications' patterns. Thus, for each application that is co-hosted with the nodes of the bursty application, it extracts the most frequent pattern, as discussed in section IIIA, that provides a sequence $S =< s_1 \ s_2 \ ... \ s_k >$ for the time window of the burst. Since the sequence arrives in order based on the timestamp we can assign each input data rate from the pattern to the input data rate R_q that we predict, for each timepoint that belongs in that window, $t \in window$. Estimating the application input rates R_q from their sequential pattern enables us to compute the components rates R_c based on the selectivity, as described above, since the input rate refers to the source component of the application. The Prediction component also requests from every component that belongs in the application graph of the bursty application to provide their $Comp_c$, CPU_c, $Comm_{c \rightarrow c+1}$, P_c, L_c, sel_c and $Deadline_q$, so as to be able to verify if the predicted rates would satisfy the system.

We select a time window, denoted as $time\_interval$, to verify the effect of the burst to the execution times of the components. The $time\_interval$ is valid from the timepoint when the burst occurred until the time interval expires. The length of the interval depends on the type of the bursts that we need to address and the criticality of the applications, and so it should be decided by the system developer with respect to the hosted applications and the expected bursts.

Afterwards, we estimate the effect of the burst to the execution times of the applications by checking the estimated rates of the components, R_c, for each timepoint t within the $time\_interval$ that we investigate and their respective measurements ($Comp_c$, $Comm_{c \rightarrow c+1}$, etc.), on the constraint formulations 3, 4 that we developed. We claim that the bursty application's rates can not be predicted, since it depicts an unexpected behavior. However, we can predict the rates for every other application, that share the same nodes with the bursty one.

Hence, if the constraints are satisfied in every examined timepoint selected, the applications can still meet the deadlines under burstiness, and we can conclude that there is no need to address the burst. However, if we compute that the forthcoming rates would lead to an infeasible solution for the constraints, that is, the applications will likely miss their deadlines, then we need to react in order to address the burst so as we maximize the probability that the deadlines of the applications are met, by triggering the burst management component. Additionally, we should trigger it when the set of the sequential patterns is incomplete for the affected applications, due to insufficient historical data. The steps of our algorithm are summarized in Algorithm 1. Note that, when multiple application graphs experience bursts, we run the prediction algorithm for each graph separately.

Algorithm 1 Prediction Algorithm

```
Define the applications $Apps_q$ that share the same nodes with the components of the bursty application
For each component $c$ that belongs in application $q$, $\forall q \in Apps_q$ request: $Comp_c$, $CPU_c$, $Comm_{c \rightarrow c+1}$, $P_c$, $L_c$, $sel_c$, $Deadline_q$
Request the sequential patterns for every $q$, $\forall q \in Apps_q$
for (Each timepoint $t$ selected within $time\_interval$) do
  Define the input rate $R_q$ for all the applications $q$, $\forall q \in Apps_q$, using their most frequent pattern
  Verify if the constraints are satisfied with the predicted input rates $R_c$, where the components rates are defined from their selectivity
  if (Constraints are not satisfied) then
    Initiate the burst management component to react to the burst and Exit
```

IV. Experimental Evaluation

A. Experimental Setup

We have implemented our technique over the Synergy middleware [1] and tested it on PlanetLab [6]. Our implementation is written in Java6 with approximately 13K lines of code. The experimental evaluation focuses on the following parameters: (i) **Behavior of our Approach**, (ii) **Application Rates**, (iii) **Throughput**, and (iv) **Overhead**. We compare our prediction approach with our **Dynamic Allocation technique** [4], that was introduced on the previous sections, to illustrate the benefit of our Prediction technique. However these approaches should work in concert, since the Dynamic Allocation should be triggered from the Prediction component whenever it identifies an infeasible rate allocation, to adjust the rates.

Traffic Monitoring Application: The experimental evaluation scenario we used was a traffic monitoring application where our goal was to identify congested areas in the Interstate 880 in California in real-time. We used Berkeley's Mobile Millennium Dataset [8], which includes real time traffic data taken from GPS-enabled phones. That dataset consists of data taken from 77 cars for the period between 10:00am and 6:00pm on the Feb 8th 2008. Each application data unit (ADU) in the dataset includes information of the form: <time, latitude, longitude, speed and carID> .

The application scenario was implemented with 4 main components: (1) A **selection component** that receives ADUs and defines the geographic region of the cars and computes each car's trajectory based on its previous location. (2) A **projection component** receives and updates the new data and projects the 50 more recent ADUs based on geographic region and trajectory information. (3) The third component computes the **average speed** for the projected ADUs to estimate the speed in the selected region. (4) The fourth component receives this data to extract the traffic result map.

Patterns: For our experiments we use 2 types of patterns, as shown in figures 3, 4, to investigate our prediction algorithm under different scenarios. In every experiment we initiate 3 applications that follow the same pattern, for the input rate of their source components, while initiating a fourth component at the same time that depicts a bursty behavior, as shown in figure 5. In both experiments we have estimated that the assigned rates will eventually render to feasible scenarios. Note that, for the units in the figures, Rate is denoted as the amount of processed ADUs per timepoint, where the Timepoint is approximately the same with the Deadline.

B. Experiments using Pattern 1

In the first set of experiments we present and evaluate the operation of our approach when the first three applications follow the first pattern, while the fourth one follows the bursty pattern. We also compare the benefit of our technique to our Dynamic Allocation technique. The applications are injected into the system with a difference of a second and their Deadline is set to 2 seconds. The experiment lasts for 5 minutes.

Prediction Rates. Figure 6 shows the rates of the applications using our Prediction approach that were processed within the Deadline. As the figure illustrates, when the rates start to increase, the applications become unable to keep up with their defined rates. We should state that each node in PlanetLab has a different processing capability and workload during the experiments. This is the reason we notice that some applications achieve higher rates than others when the nodes are congested. Thus, when the burst occurs most of the applications achieve a lower than the desired rate and especially Application 4, which is the bursty one. However, our Prediction algorithm estimates that the rates of the non-bursty applications are going to be adjusted shortly to a feasible rate, based on the application pattern. Thus, after the 60th timepoint where the rates are adjusted, every application except from the bursty one achieve their rate targets successfully. Although, Application 4 cannot achieve the maximum rate at all times, this is due to the

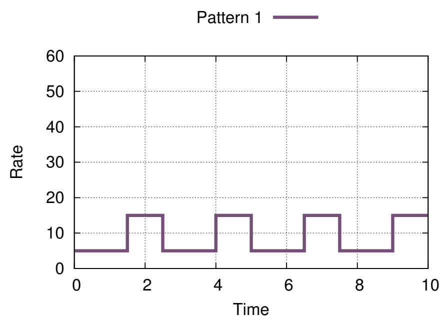

Fig. 3. Pattern 1.

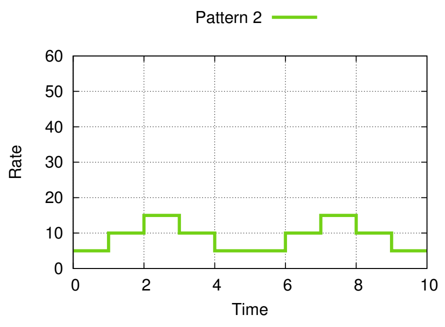

Fig. 4. Pattern 2.

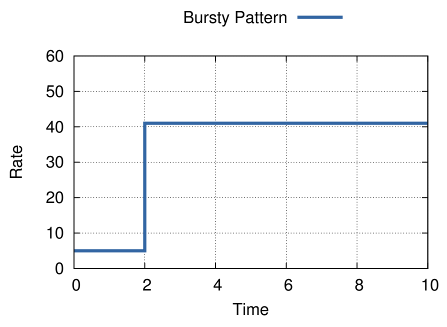

Fig. 5. Bursty Pattern

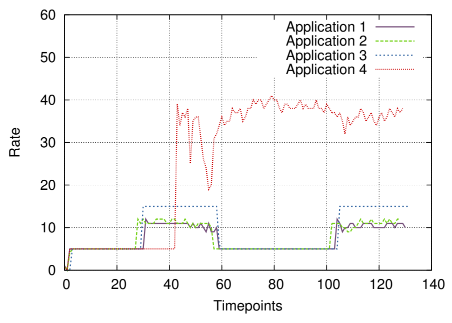

Fig. 6. Rates using Prediction.

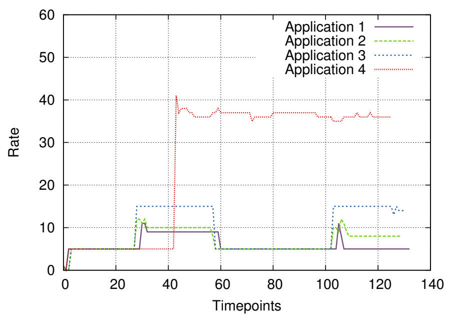

Fig. 7. Rates using Dynamic Allocation.

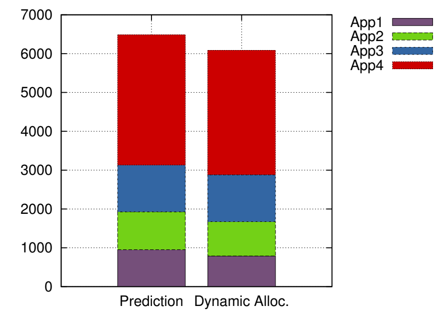

Fig. 8. Throughput - Pattern 1

unstable workload of the PlanetLab system. Then, the same behavior is being illustrated as the pattern is repeated.

Dynamic Allocation Rates. Figure 7 shows the rates of the applications using the Dynamic Allocation technique. As can be observed, using Dynamic Allocation we are able to identify the optimal rates for the current system configuration upon the onset of a burst, and so, the figure has fewer fluctuations since the rates are being adjusted to a feasible solution. Thus, after the burst has been injected after the 40th timepoint the Dynamic Allocation technique estimates the rates that would result on a stable system and assigns them. However, when the rates later decrease, due to the pattern, the system has already stabilized and so there is no need to re-adjust the rates, although this could lead to higher throughput. In addition, when the applications re-increase their rates, as the pattern dictates, the Dynamic Allocation algorithm is triggered and readjusts the system as before.

Throughput. Figure 8 illustrates the total throughput achieved in the system for both techniques, and for each application separately. As can be seen, the Prediction algorithm achieves a higher throughput, with 6486 ADUs compared to the 6086 ADUs of the Dynamic Allocation algorithm. This is basically due to the fact that the Dynamic Allocation has a lower throughput on the interval when the applications decrease their rates, since the bursty application can achieve higher rates in that period. On the other hand, the Prediction technique achieves that rate since it has identified the feasibility of that interval.

Delay. In figure 9 we can observe the average time needed for each algorithm to provide its solution. The Prediction algorithm took less than the half time to determine the feasibility of the forthcoming rates, since the Dynamic Allocation approach has to converge to a feasible solution, while the Prediction algorithm only verifies the feasibility of the solution. Moreover, an increased amount of components would raise that difference, due to the algorithms complexity. The Dynamic Allocation's delay is important since it implies queueing for the congested components and it further increases the problem until the rates are reassigned. On the other hand if the Prediction algorithm decides that there is no forthcoming feasible solution, it has to trigger the Dynamic Allocation algorithm and the outcome will be further delayed.

C. Experiments using Pattern 2

In the second set of experiments we evaluate the operation of our algorithm with the second pattern to better illustrate its behavior. The injection of the applications follows the same order as in the first set and the system configuration remains the same.

Prediction Rates. Figure 10 presents the rates of the applications over time using the Prediction approach. As the figure illustrates, the workload becomes unstable at the timepoint where the applications increase their rate to 15 and the burst is injected, and so the applications are able to process less ADUs than the target within the Deadline. However, our prediction algorithm is able to define that the workload is going to be stabilized shortly, and thus there is no need for reaction. Hence, after the rates are decreased due to the pattern all the applications achieve their target rates, except from the bursty one, for the same reason that we discussed above in pattern 1.

Dynamic Allocation Rates. Figure 11 shows the respective rates of the applications using the Dynamic Allocation technique. As can be observed, the Dynamic Allocation algorithm identifies again the optimal rates to obtain a stable rate for the current system configuration when the burst occurs, and so, the

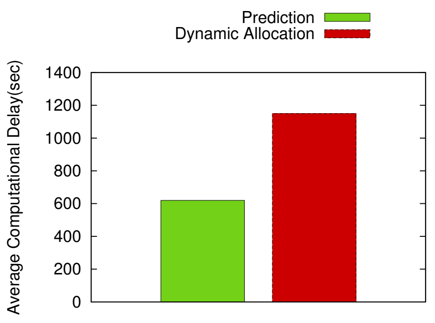

Fig. 9. Delay - Pattern 1

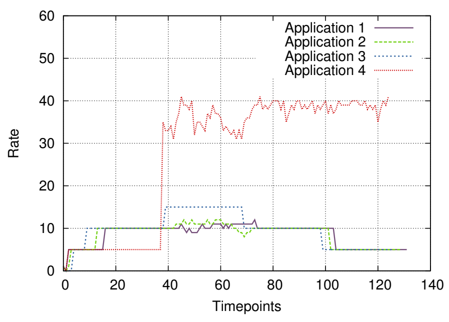

Fig. 10. Rates using Prediction.

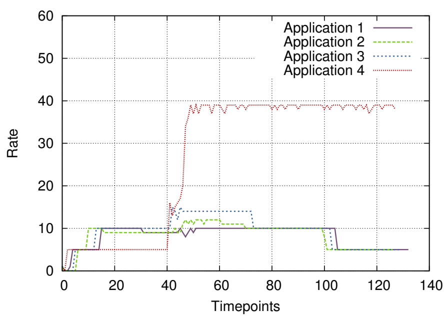

Fig. 11. Rates using Dynamic Allocation.

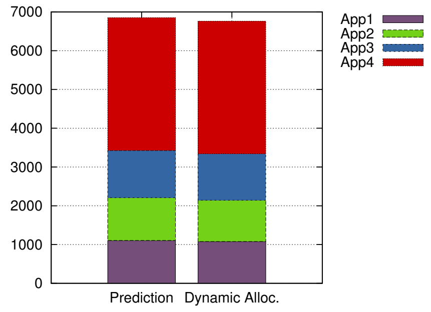

Fig. 12. Throughput - Pattern 2

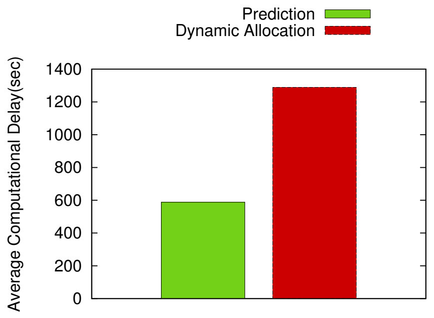

Fig. 13. Delay - Pattern 2

rate of the bursty application depicts only a few fluctuations, after the rates are being adjusted to a feasible solution.

Throughput. Figure 12 presents the total throughput of the ADUs that was achieved for each technique. The Prediction algorithm manages to achieve a higher throughput, as in the first pattern, with 6851 ADUs compared to the 6762 ADUs of the Dynamic Allocation algorithm. In this experiment the difference among the two techniques is smaller, which is attributed to the pyramid pattern. However, the advantage of the Prediction technique to achieve a higher throughput is due to its ability to identify the feasibility of the forthcoming rates rapidly, so that the rates can be maintained, when the Dynamic Allocation approach decreases them.

Delay. In figure 13 we can observe the average time needed for the execution of each algorithm on runtime. As can be observed the Prediction algorithm needs only 588 seconds compared to the 1289 seconds of the Dynamic Allocation and as we discussed above that difference would rise if we injected more components.

Discussion. From the above experiments we illustrate that the Prediction algorithm can benefit a system, when bursts occur, if the patterns of the application rates can be estimated. We have shown that the Prediction algorithm is executed faster than the burst management component and when there is no need for reaction it achieves a higher throughput. Thus, we conclude that the Prediction algorithm should be used in systems with transient bursts to define the need for reaction so as to increase the throughput and avoid the computational cost of the burst management component.

V. Related Work

Distributed stream processing systems have recently become extremely popular for processing high-throughput, low-latency data streams. A number of stream processing systems have emerged in the literature (including our own work on the Synergy middleware) [1], [14], [15]. The research in this area is very rich and many papers have been published on detailed aspects of the technology such as data models, operators and query languages, resource management, scheduling, admission control policies, load shedding, composition and placement algorithms, etc. Although, these research efforts have focused on high performance stream processing engines, our work studies the problem QoS management using pattern-based prediction under bursty conditions.

Recent efforts have studied the problem of balancing the overloads in a DSPS. In our previous work [4], [5], we have proposed the BARRE and RADAR algorithms for accommodating unpredictable bursts of the data streams in DSPS. BARRE proactively computes data stream allocations to identify all feasible allocations and uses them at runtime upon the onset of a burst. RADAR on the other hand uses only online distributed rate adaptation techniques in order to accommodate bursts. Authors in [16] also use an offline computation phase for the optimization. Their goal is to optimize the rates in distributed real-time systems. They transform the discrete rate adaptation problem to an mp-MILP problem to reduce the online computation. Lumezanu *et al* in [17], model the latency assignment problem for real-time distributed applications as a utility maximization problem. In ROD [13] operators are assigned on processing nodes in such a way that the maximum possible input rate is supported for each operator. Authors in [2] consider the problem of how to avoid overloads in distributed stream processing systems though load shedding. They propose a solution based on re-configuring and dropping data units at times of excessive load. The reconfiguration is based on redundant computations upon composition. 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 in order to reduce network delays. All of these approaches either need to know the complete system configuration in advance, to compute the feasible rate allocations or they need to time and computational resources to estimate the rate allocations when a burst occurs. On the contrary, our Prediction approach is different since it provides a lightweight solution to predict the forthcoming system workload in advance, to avoid the resource overhead of estimating the optimal rate allocations when this is possible.

The most similar problem to our approach is the work in [12]. The authors propose a QoS management scheme that features query workload estimators, that predicts the query workload using execution time profiling and adjusts the query QoS levels, using sampling, based on online query execution time prediction. However, their workload prediction is based on the current input rate of the system, while in our approach we develop a prediction scheme that is based on sequential patterns to define the input rate beforehand. Authors in [19] present a predictive resource management algorithm to achieve the timeliness requirements of periodic tasks. Their algorithm monitors the timeliness behavior of the tasks to detect the application workload changes, that may affect task timeliness, to perform resource allocation, through adaptation mechanisms such as replication of task processes or subtasks. Their technique determines the subtasks that need to be replicated along with the processor resources that are required for executing the replicas to achieve the task timeliness requirements, using a predictive technique that forecasts task timeliness and incrementally adds replicas until the forecasted timeliness are acceptable. On the other hand our technique aims to predict the forthcoming workload in advance, so as to meet the task timeliness requirements.

Several techniques can be found in the literature for identifying patterns. Our approach is able to adopt any of the techniques for mining sequential patterns to define the patterns of the applications data rates which is fundamental part on our approach. Authors in [9] propose their approach, called SPEED, to identify sequential patterns in a data stream. The originality of their technique is based on the used data structure to maintain frequent sequential patterns which is coupled with a fast pruning strategy. Ayres *et al* in [10] suggest depth-first search strategy that integrates a depth-first traversal of the search space with effective pruning mechanisms for mining sequential patterns. Their approach is especially efficient when the sequential patterns are very long. In [11], they propose an algorithm, called Moment, to identify and maintain all closed frequent itemsets in a sliding window, that contains the most recent samples in a data stream. In the Moment algorithm, they use an in-memory data structure, the closed enumeration tree to record all closed frequent itemsets in the current sliding window. In addition, the closed enumeration tree also monitors the itemsets that form the boundary between closed frequent itemsets and the rest of the itemsets.

VI. Conclusions

In this paper we have presented our approach to predict the capability of the Distributed Stream Processing System to meet the QoS requirements of the applications under bursty situations. Our approach builds application data rate patterns at run-time and predicts the effect of the burst on the performance of the applications, to identify whether the system needs to compensate delays experienced by the application components due to sudden bursts of load. Our detailed experimental results over the Synergy middleware illustrate that our approach is practical, depicts good performance and has low overhead.

Acknowledgment

This research has been co-financed by the European Union (European Social Fund ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program:Thalis-DISFER, Aristeia-MMD Investing in knowledge society through the European Social Fund and the FP7 INSIGHT project.

References

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

[2] 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.

[3] C. Lu, X. Wang, and X. Koutsoukos, "Feedback utilization control in distributed real-time systems with end-to-end tasks," *IEEE TPDS*, vol. 16, no. 6, pp. 550–561, Jun 2005.

[4] I. Boutsis and V. Kalogeraki, "Radar: Adaptive rate allocation in distributed stream processing systems under bursty workloads," in *SRDS*, Irvine, California, October 2012.

[5] Y. Drougas and V. Kalogeraki, "Accommodating bursts in distributed stream processing systems," in *IPDPS*, Rome, Italy, May 2009.

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

[7] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object address and routing for large-scale peer-to-peer systems," in *IFIP/ACM Middleware, Heidelberg, Germany*, November 2001.

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

[9] C. Raïssi, P. Poncelet, and M. Teisseire, "Need for speed : Mining sequential patterns in data streams," in *BDA*, Saint Malo, October 2005.

[10] J. Ayres, J. Flannick, J. Gehrke, and T. Yiu, "Sequential pattern mining using a bitmap representation," in *KDD*, Edmonton, Alberta, Canada, July 2002.

[11] Y. Chi, H. Wang, P. S. Yu, and R. R. Muntz, "Moment: Maintaining closed frequent itemsets over a stream sliding window," in *In ICDM*, Brighton, UK, November 2004, pp. 59–66.

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

[13] Y. Xing, J.-H. Hwang, U. Çetintemel, and S. Zdonik, "Providing resiliency to load variations in distributed stream processing," in *Proc. of VLDB*, Seoul, Korea, Sep. 2006.

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

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

[16] 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.

[17] 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.

[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.

[19] B. Ravindran and T. Hegazy, "A predictive algorithm for adaptive resource management of periodic tasks in asynchronous real-time distributed systems," in *IPDPS*, San Francisco, California, April 2001.