QoS-Aware Shared Component Composition for Distributed Stream Processing Systems

Thomas Repantis, *Student Member, IEEE,* Xiaohui Gu, *Member, IEEE,* and Vana Kalogeraki, *Member, IEEE*

***Abstract*—Many emerging on-line data analysis applications require applying continuous query operations such as correlation, aggregation, and filtering to data streams in real-time. Distributed stream processing systems allow in-network stream processing to achieve better scalability and quality-of-service (QoS) provision. In this paper we present *Synergy*, a novel distributed stream processing middleware that provides automatic *sharing-aware component composition* capability. Synergy enables efficient reuse of both result streams and processing components, while composing distributed stream processing applications with QoS demands. It provides a set of fully distributed algorithms to discover and evaluate the reusability of available result streams and processing components when instantiating new stream applications. Specifically, Synergy performs *QoS impact projection* to examine whether the shared processing can cause QoS violations on currently running applications. The QoS impact projection algorithm can handle different types of streams including both regular traffic and bursty traffic. If no existing processing components can be reused, Synergy dynamically deploys new components at strategic locations to satisfy new application requests. We have implemented a prototype of the Synergy middleware and evaluated its performance on both PlanetLab and simulation testbeds. The experimental results show that Synergy can achieve much better resource utilization and QoS provisioning than previously proposed schemes, by judiciously sharing streams and components during application composition.**

***Index Terms*—Distributed Stream Processing, Component Composition, Shared Processing, QoS, Resource Management.**

I. Introduction

Stream processing applications have gained considerable acceptance over the past few years in a wide range of emerging domains such as monitoring of network traffic for intrusion detection, surveillance of financial trades for fraud detection, observation of customer clicks for e-commerce applications, customization of multimedia or news feeds, and analysis of sensor data in real-time [1], [2]. Streams are sequences of data tuples arriving continuously, which need to be processed in real-time to generate outputs of interest or to identify meaningful events. Streams are processed by components; each component represents a processing element that operates on a set of input streams to produce a set of output streams. Stream processing applications are instantiated as directed acyclic graphs connecting components. Two components are connected if the output of one becomes the input of the other to accomplish the application execution. Often, the data sources, as well as the components that implement the application logic are distributed across multiple machines connected through a network, constituting a distributed stream processing system (DSPS) (e.g., [3]–[8]).

The IBM System S reference application [9] provides a concrete stream processing application example. It is a multi-modal stream analytic and monitoring application for the processing of claims related to disaster assistance. In a disaster claim processing center, agents receive phone calls from applicants to process claims. Correlated, multi-modal source streams are then generated while a claim is being processed. These may include data records, phone conversations, and e-mail communication. The goal of this stream processing application is to prevent fraudulent or unfairly treated claims, as well as to identify problematic agents and their accomplices. This is achieved by processing streams such as the conversations between agents and applicants, e-mail logs, video from the processing center, and selective news feeds. More than 50 components with a variety of computational requirements constitute the graph describing the application. The stream processing components include load diffusers, decision trees, joins, top-k selectors, and index builders. For example, a component labels an agent as problematic by searching for keywords in their conversations with applicants, while another component computes a suspicion level for an applicant based on the amount they claim and personal characteristics such as their income. These components process a variety of streams, ranging from text to audio and video, which have extremely different rates. Stream sources often produce large volumes of data at high rates, while workload spikes cannot be predicted in advance. Providing low-latency, high-throughput execution for such distributed applications entails considerable strain on both communication and processing resources and thus presents significant challenges to the design of a DSPS.

While a DSPS provides the components that are needed to develop and execute an application, a major challenge remains: How to select among different component instances to compose stream processing applications on-demand. While previous efforts have investigated several aspects of component composition [5], [6] and placement [7] for stream applications, our research focuses on enabling *sharing-aware component composition* for efficient distributed stream processing. Sharing-aware composition allows different applications to utilize previously generated streams and already deployed stream processing components. The distinct characteristics of distributed stream processing applications make sharing-aware component composition particularly challenging. First,

T. Repantis and V. Kalogeraki are with the Department of Computer Science & Engineering, University of California, Riverside, CA, 92521. Email: {trep,vana}@cs.ucr.edu

X. Gu is with the Department of Computer Science, North Carolina State University, Raleigh, NC 27695. Email: gu@csc.ncsu.edu

stream processing applications often have minimum quality-of-service (QoS) requirements (e.g., end-to-end delay). In a shared processing environment, the QoS of a stream processing application can be affected by multiple components that are invoked concurrently and asynchronously by many applications. Second, stream processing applications operate autonomously in a highly dynamic environment, with load spikes and unpredictable occurrences of events. Thus, composition must be performed quickly, during run-time, and must be able to adapt to dynamic traffic changes, including bursts. Third, congruent with related efforts [5]–[9], we expect distributed solutions to be more appropriate for federated DSPSs that scale to thousands of streams, components, and nodes. This is also supported by the analytical and experimental comparison between centralized and distributed composition algorithms provided in [10]. The overhead comparison therein indicates that the relative merit between distributed and centralized solutions is decided by the size of the overlay network, the overlay topology, the number of stream processing components, the application request rate, and the frequency with which state updates have to be communicated to other nodes. The global state of a large-scale DSPS is changing faster than it can be communicated to a single host. This renders it challenging for a single host to make accurate global decisions when large numbers of nodes and applications are involved.

Despite the aforementioned challenges, there are significant benefits to be gained from a flexible sharing-aware component composition: i) *enhanced QoS provisioning* (e.g., shorter service delay) since existing streams that meet the user's requirements can be furnished immediately, while the time-consuming process of new component deployment is triggered only when none of the existing components can accommodate a new request; and ii) *reduced resource load* for the system, by avoiding redundant computations and data transfers. This results in a significant improvement in the performance and scalability of the entire system.

In this paper we present *Synergy*, a distributed stream processing middleware that provides sharing-aware component composition. Synergy is implemented on top of a wide-area overlay network and undertakes the composition of distributed stream processing applications. Synergy supports both data stream and processing component reuse while ensuring that the application QoS requirements can be met. The decision of which components or streams to reuse is made dynamically at run-time taking into account the applications' QoS requirements and the current system resource availability. Specifically, this paper makes the following major contributions:

–We propose a decentralized light-weight composition algorithm that discovers streams and components at run-time and checks whether any of the existing components or streams can satisfy the application's request. After the qualified candidate components have been identified, components and streams are selected and composed dynamically to meet the application resource and QoS requirements.

–We integrate a QoS impact projection mechanism into the distributed component composition algorithm to evaluate the reusability of existing stream processing components according to the applications' QoS constraints. When a component is shared by multiple applications, the QoS of each application that uses the component may be affected due to increased queueing delays on the processors and the communication links. Synergy's approach is to predict the impact of the additional workload on the QoS of the affected applications and ensure that a component reuse does not cause QoS violations in existing stream applications. Such a projection can facilitate QoS provisioning for both the newly admitted and the current applications. Our projection algorithm considers not only regular but also bursty stream traffic [11] such as voice-over-IP streams, network traffic and sensor data streams.

–Synergy dynamically deploys new components at strategic locations to satisfy new application requests. Component deployment is triggered when a requested component does not exist, or when none of the existing components can safely provide the requested stream processing due to resource overloads or QoS violations.

–We have implemented a prototype of Synergy and evaluated its performance on the PlanetLab [12] wide-area network testbed. We have also conducted extensive simulations to compare Synergy's composition algorithm to existing alternative schemes. The experimental results show that: i) Synergy consistently achieves much better QoS provisioning compared to other approaches, for a variety of application loads, ii) sharing-aware component composition increases the number of admitted applications, while scaling to large request loads and network sizes, iii) QoS impact projection greatly increases the percentage of admitted applications that meet their QoS requirements, iv) the QoS impact projection algorithm shows good prediction accuracy for both regular and bursty stream traffic, and v) Synergy's decentralized composition protocol has low message overhead and offers minimal setup time, in the order of a few seconds.

II. System Model

A. Stream Processing Application Model

Table 1 summarizes the notations we use while discussing our model. A data stream s_j consists of a sequence of continuous data tuples. A stream processing component c_i is defined as a self-contained processing element that implements an atomic stream processing operator o_i on a set of input streams $\{is_i\}$ and produces a set of output streams $\{os_i\}$. Stream processing components can have more than one input (e.g., a join operator) and outputs (e.g., a split operator). Each atomic operator can be provided by multiple components $c_1, \ldots, c_k$, which are essentially multiple instances of the same operator. We associate metadata with each deployed component or existing data stream in the system to facilitate the discovery process. Both components and streams are named based on a common ontology (e.g., o_i.name = Aggregator.COUNT, s_j.name = Video.MPEGII.EntranceCamera). The name of a stream produced by a source is given based on the ontology and may incorporate the source node characteristics (e.g., IP and port), if these affect the semantics of the stream. As streams are processed by components, their names reflect the stream processing operators that have been applied to them. For example, in Figure 5, the name of s_2 is $o_1(s_1)$, to reflect

Notation	Meaning
c_i	Component
o_i	Operator
l_j	Virtual Link
s_j	Stream
ξ	Query Plan
λ	Application Component Graph
Q_ξ	End-to-End QoS Requirements
Q_λ	End-to-End QoS Achievements
p_{v_i}	Processor Load on Node v_i
b_{l_j}	Network Load on Virtual Link l_j
rp_{v_i}	Residual Processing Capacity on Node v_i
rb_{l_j}	Residual Network Bandwidth on Virtual Link l_j
τ_{c_i}	Processing Time for c_i
x_{c_i,v_i}	Mean Execution Time for c_i on v_i
σ_{s_j}	Transmission Time for s_j
y_{s_j,l_j}	Mean Communication Time for s_j on l_j
q_t	Requested End-to-End Execution Time
$\bar{t}$	Projected End-to-End Execution Time
p_{o_i}	Processing Time Required for o_i
b_{s_j}	Bandwidth Required for s_j

Fig. 1. Notations.

that s_2 is the output of operator o_1 on the input stream s_1. Similarly, the name of s_4 is $o_2(o_1(s_1))$.

A stream processing request (query) is described by a given *query plan*, denoted by ξ. The query plan is represented by a directed acyclic graph (DAG) specifying the required operators o_i and the streams s_j among them. A query plan can be dynamically instantiated into different *application component graphs*, denoted by λ, depending on the processing and bandwidth availability. The vertices of an application component graph represent the components being invoked at a set of nodes to accomplish the application execution, while the edges represent virtual network links between the components, each one of which may span multiple physical network links. An edge connects two components c_i and c_j if the output of the component c_i is the input for the component c_j. The application component graph is generated by Synergy's component composition algorithm at run-time, after selecting among different component candidates that provide the required stream processing operators o_i and satisfy the end-to-end QoS requirements Q_ξ. Synergy's component composition algorithm is described in Section III-A.

B. QoS Model

A query plan ξ, describing a stream processing request, includes the processing requirements of the requested operators $p_{o_i}, \forall o_i \in \xi$ and the bandwidth requirements of the corresponding streams $b_{s_j}, \forall s_j \in \xi$. The bandwidth requirements are calculated according to the user-requested stream rate, while the processing requirements are calculated according to the data rate and profiled processing times for the operators [13]. The stream processing request also specifies the end-to-end requirements Q_ξ, for m different QoS metrics such as end-to-end execution time and loss rate, $Q_\xi = [q_1, ...q_m]$. Although our schemes are generic to additive QoS metrics, we focus on end-to-end execution time, denoted by q_t, which is computed as the sum of the processing and communication times for a data tuple to traverse the whole query plan.

After admitting an application request, the residual processing capacity on every node v_i participating in the application execution must be $rp_{v_i} \geq 0$. Similarly, the residual available bandwidth on each virtual link l_j connected to each v_i must be $rb_{l_j} \geq 0$. Finally, the end-to-end QoS requirements specified in the query plan ξ must be met by the final application component graph λ, i.e., $q_t^\lambda \leq q_t^\xi$.

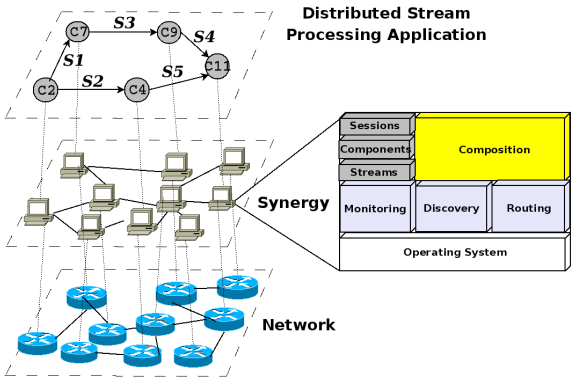

Fig. 2. Synergy system architecture.

C. Synergy Architecture

Synergy is a wide-area stream processing middleware that consists of a set of distributed hosts v_i connected via virtual links l_j, that create an overlay mesh on top of the existing IP network. Figure 2 shows an overview of the architecture. Synergy leverages the routing layer of the underlying overlay network for registering and discovering available components and streams in a decentralized manner. Synergy adopts a fully distributed architecture, where any node of the middleware can compose a distributed stream processing application.

Each Synergy node, denoted by v_i, is identified by its IP and port. As illustrated in Figure 2, each node maintains a *metadata repository* of active stream processing sessions, streams, and components (including input and output buffers). Additionally, the architecture of a Synergy node includes the following main modules: i) a *composition module* that is responsible for running the component composition algorithm and uses: ii) a *discovery module* that is responsible for locating existing data streams and components. In our current Synergy prototype we implement a keyword-based discovery service [14] on top of the Pastry distributed hash table (DHT) [15]. However, our middleware can also be integrated with other DHTs, or unstructured overlays [16], since discovery is an independent module of our system. iii) a *routing module* that routes data streams between different Synergy nodes; and iv) a *monitoring module* that is responsible for maintaining resource utilization information for v_i and the virtual links connected to v_i. The monitoring module keeps track of the CPU load and network bandwidth. The current processor load p_{v_i} and the residual processing capacity rp_{v_i} on node v_i are inferred from the CPU idle time as measured from the `/proc` interface. The residual available bandwidth rb_{l_j} on each virtual link l_j connected to v_i is measured using a bandwidth measuring tool (e.g., Iperf). We finally use b_{l_j} to denote the amount of current bandwidth consumed on l_j.

D. Approach Overview

A stream processing application request is submitted directly to a Synergy node v_s, if the client is running the middleware, or redirected to a Synergy node v_s that is closest to the client based on a predefined proximity metric (e.g., geographical location). Alternative policies can select v_s to be the Synergy node closest to the source or the sink node(s) of

the application. The user submits a query plan ξ, that specifies the required operators and the order in which they will execute. The processing requirements of the operators $p_{o_i}, \forall o_i \in \xi$ and the bandwidth requirements of the streams $b_{s_j}, \forall s_j \in \xi$ are also included in ξ. The request also specifies the end-to-end QoS requirements $Q_\xi = [q_1, ...q_m]$ for the composed stream processing application. These requirements (i.e., ξ, Q_ξ) are used by the Synergy middleware running on that node to initiate the distributed component composition protocol. This protocol produces the application component graph λ that identifies the particular components that shall be invoked to instantiate the new request.

To avoid redundant computations, Synergy first tries to discover whether any of the requested streams have been generated by previously instantiated query plans. To maximize the sharing benefit, Synergy reuses the result stream(s) generated during the latest possible stages in the query plan. Thus, only the remaining operators in the query plan are needed to generate the user requested stream(s). Synergy then probes the candidate nodes that can provide these operators, to determine: i) whether they have the available resources to accommodate the new application, ii) whether the end-to-end delay is within the required QoS, and iii) whether the impact of the new application would cause QoS violations to existing applications. During the probing process, the system may need to decide where to deploy new processing components. Deployment takes place if none of the existing components can provide a requested stream processing operator, or if there exist such components, but none of them can be safely reused without resource overloads or QoS violations. Synergy adopts a collocation-based component deployment strategy to minimize the number of hops that streams travel through.

Figure 3 gives a very simple example of how probes can be propagated hop-by-hop to test many different component combinations. Assuming components c_1 and c_2 offer operator o_1, while components c_3 and c_4 offer operator o_2, and assuming that the components can be located at any node in the system, probes will attempt to travel from the source S to the destination D through paths $S \rightarrow c_1 \rightarrow c_3 \rightarrow D$, $S \rightarrow c_1 \rightarrow c_4 \rightarrow D$, $S \rightarrow c_2 \rightarrow c_3 \rightarrow D$, and $S \rightarrow c_2 \rightarrow c_4 \rightarrow D$. A probe is dropped in the middle of the path if any of the above conditions are not satisfied in any hop. Thus, the paths that create resource overloads, result to end-to-end delays outside the requested QoS limits, or unacceptably increase the delays of the existing applications, are eliminated. From the successful candidate application component graphs, our composition algorithm selects the one that results in a more balanced load in the system and the new stream application is instantiated.

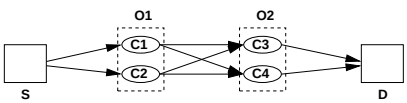

Fig. 3. Probing example.

III. Design and Algorithm

A. Synergy Component Composition Protocol

Synergy's fully distributed composition protocol is executed when instantiating a new application. Given a stream processing request, a Synergy node first gets the locally generated query plan ξ and then instantiates the application component graph based on the user's QoS requirements Q_ξ. Figure 5 shows an example of a query plan, while Figure 6 shows a corresponding component composition example. To achieve decentralized, light-weight component selection, Synergy employs a set of probes to concurrently discover and select the best composition. Synergy differs from previous work (e.g., [5], [14]) in that it judiciously considers the impact of stream and component sharing on both the new and existing applications. The probes carry the original request information (i.e., ξ, Q_ξ), collect resource and QoS information from the distributed components, perform QoS impact projection, and select qualified compositions according to the user's QoS requirements. The best composition is then selected among all qualified ones, based on a load balancing metric. The composition protocol, a high level description of which is shown in Algorithm 4, consists of five main steps:

Step 1. Probe creation. Given a stream processing query plan ξ, the Synergy node v_s first discovers whether any existing streams can be used to satisfy the user's request. The goal is to reuse existing streams as much as possible to avoid redundant computations. For example, in Figure 5, starting from the destination, v_s will first check if the result stream (s_8) is available. If not, it will look for the streams one hop away from the destination (s_6 and s_7), then two hops away from the destination (s_4 and s_5) and so on, until it can find any streams that can be reused. We denote this Breadth First Search on the query plan as identification of the *maximum sharable point(s)*. The nodes generating the reusable streams may not have enough available bandwidth for more streaming sessions or may have virtual links with unacceptable communication latencies. In that case all probes are dropped by those nodes and v_s checks whether there exist components that can provide the operators requested in the query plan, as if no streams had been discovered. The details about determining the maximum sharable points and about discovering sharable streams and components are described in Section III-C. Next, the Synergy node v_s initiates a distributed probing process to collect resource and QoS states from those candidate components that provide the maximum sharable points. The goal of the probing process is to select qualified candidate components that can best satisfy ξ and Q_ξ and result in the most balanced load in the system. The initial probing message carries the request information (ξ and Q_ξ) and a probing ratio, that limits the probing overhead by specifying the maximum percentage of candidate components that can be probed for each required operator. The probing ratio can be statically defined, or dynamically decided by the system, based on the operator, the components' availability, the user's QoS requirements, current conditions, or historical measurement data [5]. The initial probing message is sent to the nodes hosting components offering the maximum sharable points. We do not probe the nodes that are generating streams before the maximum sharable points, since the overhead would be disproportional to the probability that they can offer a better component graph in terms of QoS.

Input: query $\langle \xi, Q_\xi, \rangle$, node v_s
Output: application component graph λ
v_s identifies *maximum sharable point(s)* in ξ
v_s spawns initial probes
for each v_i in path
  checks available resources
  AND checks QoS so far in Q_ξ
  AND checks *projected QoS impact*
  if probed composition qualifies
    sends acknowledgement message to upstream node
    performs transient resource reservation at v_i
    discovers next-hop candidate components from ξ
    deploys next-hop candidate components if needed
    spawns probes for selected components
  else drops received probe
v_s selects most load-balanced component composition λ
v_s establishes stream processing session

Fig. 4. Synergy composition algorithm.

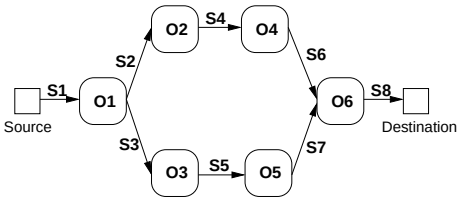

Fig. 5. Query plan example.

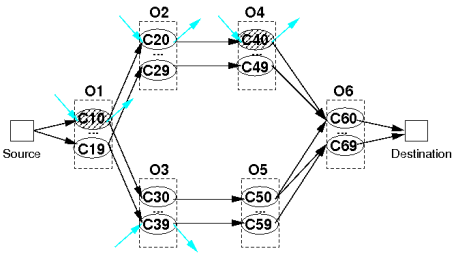

Fig. 6. Synergy composition example.

Step 2. Probe processing. When a Synergy node v_i receives a probing message called probe P_i, it processes the probe based on its local state and on the information carried by P_i. A probe has to satisfy three conditions to qualify for further propagation: First, v_i calculates whether the requested processing and bandwidth requirements p_{o_i} and b_{s_j} can be satisfied by the available residual processing capacity and bandwidth rp_{v_i} and rb_{l_j}, of the node hosting the component and of the virtual link the probe came from respectively. Thus, both $rp_{v_i} \geq p_{o_i}$ and $rb_{l_j} \geq b_{s_j}$ have to hold. Second, v_i calculates whether the QoS values of the part of the component graph that has been probed so far already violate the required QoS values specified in Q_ξ. For the end-to-end execution time QoS metric q_t this is done as follows: The sum of the components' processing and transmission times so far has to be less than q_t. The time that was needed for the probe to travel so far gives an estimate of the transmission times, while the processing times are estimated in advance from profiling [13]. Third, v_i calculates the QoS impact on the existing stream processing sessions by admitting this new request. In particular, the expected execution delay increase due to the additional stream volume introduced by the new request is calculated. The details about the QoS impact projection are described in Section III-D. Similarly, the impact of the existing stream processing sessions on the QoS of the new request is calculated. Both the new and the existing sessions have to remain within their QoS requirements.

If any of the above three conditions cannot be met, the probe is dropped immediately to reduce the overhead. Otherwise, the node sends an acknowledgement message to its upstream node, and performs *transient* resource reservation to avoid overbooking due to concurrent probes for different requests. The transient resource reservation is cancelled after a timeout period if the node does not receive a confirmation message to setup the stream processing application session.

Step 3. Hop-by-hop probe propagation. If the probe P_i has not been dropped, v_i propagates it further. v_i derives the next-hop operators from the query plan and acquires the locations of all available candidate components for each next-hop operator using the discovery module of the middleware. Then v_i selects a number of candidate components to probe, based on the probing ratio. If more candidates than the number specified by the probing ratio are available, random ones are selected, or –if a delay monitoring service [17] is available– the ones with the smallest communication delay are selected. If no candidate components for the next operator are found, or if no candidate components return acknowledgement messages, a new component is deployed, following the protocol described in Section III-B. The deployment protocol aims at collocating the new component with either its upstream or its downstream component in the query plan, in order to minimize the number of hops that streams have to travel through.

After the candidate components have been selected, v_i *spawns* new probes from P_i to all selected next-hop candidates. Each new probe, in addition to ξ (including p_{o_i} and b_{s_j}), Q_ξ, and the probing ratio, carries the up-to-date resource state of v_i, namely rp_{v_i} and rb_{l_j}, and of all the nodes the previous probes have visited so far. Finally, v_i sends all new probes to the nodes hosting the selected next-hop components.

A protocol optimization to reduce probing could involve piggybacking load and application QoS information on streaming data. This way nodes that are hosting applications could inform their downstream nodes regarding their current state and would not need to be probed by them.

Step 4. Composition selection. After reaching the destination specified in ξ, all successful probes belonging to a composition request return to the original Synergy node v_s that initiated the probing protocol. After selecting all qualified candidate components, v_s first generates complete candidate component graphs from the probed paths. Since the query plan is a DAG, v_s can derive complete component graphs by merging the probed paths. For example, in Figure 6, a probe can traverse $c_{10} \rightarrow c_{20} \rightarrow c_{40} \rightarrow c_{60}$ or $c_{10} \rightarrow c_{30} \rightarrow c_{50} \rightarrow c_{60}$. Thus, v_s merges these two paths into a complete component graph. Second, v_s calculates the requested and residual resources for the candidate component graphs based on the precise states collected by the probes. Third, v_s selects qualified compositions according to the user's operator, resource, and QoS requirements. Let V_λ be the set of nodes that is being used to instantiate λ. We use $c_i.o$ to represent the operator provided by the component c_i. The selection conditions are as follows:

$$operator \quad constraints : c_i.o = o_i, \ \forall o_i \in \xi, \exists c_i \in \lambda \quad (1)$$

$$QoS \quad constraints : q_r^\lambda \leq q_r^\xi, 1 \leq r \leq m \quad (2)$$

$$processing \quad capacity \quad constraints : rp_{v_i} \geq 0, \forall v_i \in V_\lambda \quad (3)$$

$$bandwidth \quad constraints : rb_{l_j} \geq 0, \quad \forall l_j \in \lambda \quad (4)$$

Among all the qualified compositions that satisfy the application QoS requirements, v_s selects the best one according to the following load balancing metric $\phi(\lambda)$. The qualified composition with the smallest $\phi(\lambda)$ value is selected:

$$\phi(\lambda) = \sum_{v_i \in V_\lambda, o_i \in \xi} \frac{p_{o_i}}{rp_{v_i} + p_{o_i}} + \sum_{l_j \in \lambda, s_j \in \xi} \frac{b_{s_j}}{rb_{l_j} + b_{s_j}} \quad (5)$$

Step 5. Application session setup. Finally, the Synergy node v_s establishes the stream processing application session by sending confirmation messages along the selected application component graph. If no qualified composition can be found (i.e., all probes were dropped, including the ones without stream reuse), then the existing components and nodes in the probing path are too overloaded. Thus, these nodes cannot accommodate the requested application with the specified QoS requirements, or host new components. v_s can then try to deploy a new complete application component graph in strategically chosen places in the network [7], [18]. The goal of the described protocol is to discover and select existing streams and components to share, in order to accommodate a new application request, assuming components are already deployed on nodes. This is orthogonal to the policies that might be in place regarding the deployment of new complete application component graphs, which is outside the scope of this paper. If deploying a new complete application component graph also fails, v_s returns a failure message.

Synergy is adaptable middleware, taking into account the current status of the dynamic system at the moment the application request arrives. Therefore, it does not compare to optimal solutions calculated offline that apply to static environments. Furthermore, Synergy decides the admission of applications depending on whether QoS can be *fully* met or not. Statistical methods [14] could be adopted to integrate our solution with utility-based approaches [16], in which case different *levels* of QoS would be offered. In that case, QoS requirements can be expressed as satisfaction probabilities, and histograms can be maintained to calculate the probabilities of dynamic resource availability. Different weights can be assigned to different applications based on their importance, determining the probing ratio, as well as the maximum QoS level of particular applications. The system would then decide the probability with which a certain application could be provided with the maximum possible QoS level.

B. New Component Deployment

New component deployment is triggered when i) no candidate components for a requested operator are returned by the peer-to-peer overlay, or ii) when candidate components exist, but none of them can be safely reused. This can be the case if sharing the existing components would cause resource overloads, or QoS violations to the new or to existing applications. Each node processing a probe requires each next-hop candidate component to send an acknowledgement message back if the probe conditions can be satisfied. The node initiates a new component deployment if it does not receive any acknowledgement message from its next-hop candidates.

We choose to *collocate* the new component with either its upstream or its downstream component, as this approach minimizes the number of hops in the application component graph. If collocation with an upstream component is decided, it occurs at the node that just processed a probe. If collocation with a downstream component is decided, it happens at the node a probe is forwarded to after being processed. We now discuss how the nodes to host a new component are chosen and then we describe how component deployment takes place.

Depending on the position of the missing component in the application component graph, we distinguish between three different cases, shown in Figure 7: i) If the missing component is at the beginning of the graph, we can collocate it with any of its downstream candidates. Thus, in Figure 7.a), the missing component for operator o_1 can be collocated with c_{21}, c_{22}, or c_{31}. ii) If the missing component is at the end of the graph, we can collocate it with any of its upstream candidates. Thus, in Figure 7.b), the missing component for operator o_6 can be collocated with c_{41}, c_{51}, or c_{52}. iii) If the missing component is in the middle of the graph, we can collocate it with any of its downstream or upstream candidates. Thus, in Figure 7.c), the missing component for operator o_4 can be collocated with c_{21}, c_{22}, or c_{61}. Our goal when trying to decide whether to collocate with downstream or with upstream candidates is to reduce network traffic. To that effect, we choose whether to collocate with an upstream or a downstream candidate based on the operator's profiled selectivity [19], which is included in the query plan ξ. The selectivity of an operator is calculated as the ratio of the size of its output streams over the size of its input streams, during the period of time the profiling occurs. The selectivity of an operator can be less than one, e.g., for a filter, equal to one, e.g., for a sort, or even greater than one, e.g., for some cases of a join. For selectivity less than or equal to one we collocate with an upstream candidate, while for selectivity greater than one we collocate with a downstream candidate component. Thus, the network traffic across the components is minimized. For example, in Figure 7.c), if the selectivity of the operator o_4 is less than or equal to one, we collocate the missing component for o_4 with one of the upstream candidates c_{21} or c_{22}. If on the other hand the selectivity of the operator o_4 is greater than one, we collocate the missing component with its downstream candidate c_{61}.

Since, at each hop, many probes are spawned before the final composition selection, many alternative deployments for a new component may exist. For example, in Figure 7.c), if the components that offer the operator o_4 are missing, the deployment alternatives may include the nodes hosting each of the components c_{21} and c_{22}. The resource and QoS checks described in step 2 of the composition protocol are performed on the tentatively deployed components as well. Thus a probe is dropped if resource or QoS violations are detected.

Depending on resource availability, the upstream or downstream candidates may not be able to deploy the requested component. In that case, the node that initiated the component deployment does not receive any acknowledgement message, and tries to identify other candidates. If the missing component is in the middle of the graph, both downstream and upstream candidates can be probed. If the extra candidates drop the probe as well, overlay neighbors or nodes along the probing path so far can be used. If none of these cases, for any of the

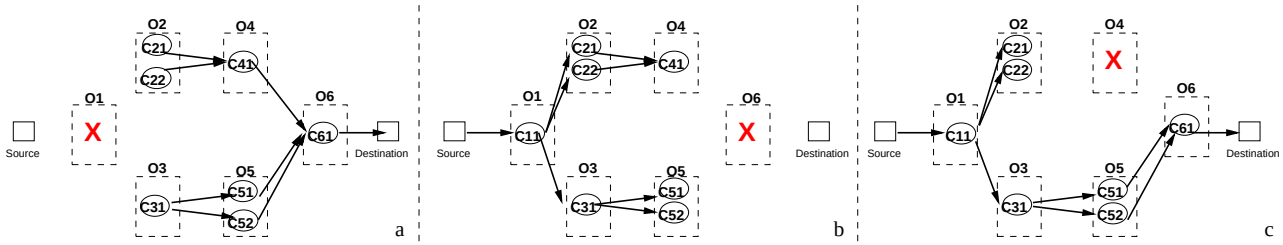

Fig. 7. The three cases of new component deployment.

probing paths, results to a successful deployment, a complete graph, as described in protocol step 5, can be deployed.

If the resource availability of a node allows the new component deployment, a transient resource reservation for this component takes place. Thus, resources are reserved, to avoid overbooking by concurrent probing processes, but the component is only tentatively deployed. After the final component graph is selected, the new components that are included in that graph are actually deployed. The rest of the transient resource reservations made by the tentatively deployed components timeout, which frees the resources for future requests. Only the permanently deployed components register their metadata with the peer-to-peer overlay, to enable their discovery and reuse by other applications.

While the collocation-based component deployment strategy minimizes the number of hops that streams travel through, it does not necessarily provide the minimum end-to-end application delay. The reason is that the triangle inequality does not necessarily hold for all nodes in real-world, large-scale distributed systems [17]. For example, in Figure 7.c), a node to host c_{41} may exist, such that the delay $c_{21} \rightarrow c_{41} \rightarrow c_{61}$ is smaller than the delay $c_{21} \rightarrow c_{61}$. However, examining all nodes for all alternative probes, would lead to an explosion of combinations. Yet, while our minimum hop component deployment does not necessarily produce the optimal solution, it heuristically provides us with several good alternatives that satisfy the QoS of the application.

C. Maximum Stream Sharing

Synergy utilizes a peer-to-peer discovery module for registering and discovering the available components and streams in a decentralized manner. As was mentioned in Section II-C, the current Synergy implementation is built over Pastry [15]. We follow a simple approach to enable the storage and retrieval of the static metadata of components and streams in the DHT, which include the location (node) hosting the component or stream. As was described in Section II-A, each component and stream is given a name, based on a common ontology. This name is converted to a key, by applying a secure hash function (SHA-1) on it, whenever a component or stream needs to be registered or discovered. On the DHT this key is used to map the metadata to a specific node, with the metadata of multiple components offering the same operator, or multiple streams carrying the same data, being stored in the same node. Configuration changes caused by node arrivals and departures are handled gracefully by the DHT. Whenever components are deployed or deleted, or streams are generated by new application sessions, or removed because they are not used by any sessions anymore, the nodes hosting them register or unregister their metadata with the DHT.

The stream processing query plan ξ specifies the operators o_i and streams s_j needed for the application execution. Using a *Maximum Sharing Discovery algorithm*, the Synergy node in which the query plan was submitted utilizes the peer-to-peer overlay for discovering existing streams and components. Since different users can submit queries that have the same or partially the same query plans, we want to reuse existing streams as much as possible to avoid redundant computations. The goal of the Maximum Sharing Discovery algorithm is to identify the *maximum sharable point(s)* in ξ. This is the operator(s) closest to the destination (in terms of hops in ξ), whose output streams currently exist in the system and can (at least partially) satisfy the user's requirements. An extreme case is that the final stream or streams already exist in the system, which can then be returned to the user directly without any further computation, as long as the residual bandwidth and communication latencies permit so. For example in Figure 5 if s_8 is already available in the system, it can be reused to satisfy the new query, incurring only extra communication but no extra processing overhead. In that case, the maximum sharable point in ξ is o_6 and Synergy will prefer to use no components if possible. If the final stream or streams are not available, the Synergy node *backtracks* hop-by-hop the query plan to find whether preceding intermediate result streams exist. For example, in Figure 5, if result streams s_8 and s_7 are not found, but s_6 and s_5 are already available in the system, they may be reused to satisfy part of the query plan. By reusing these existing streams, the Synergy node will prefer to compose a partial component graph covering the operators after the reused streams, if the resource and QoS constraints permit so. In that case, the maximum sharable points in ξ are o_3 and o_4 and only components offering operators o_5 and o_6 will be needed. To discover existing streams and existing components the peer-to-peer overlay is utilized as was described above.

D. QoS-Aware Component Sharing

To determine whether an existing candidate component can be reused to satisfy a new request, we estimate the impact of the component reuse on the latencies of the existing applications. An existing component can be reused if the additional workload brought by the new application will not violate the QoS requirements of the existing stream processing

applications (and similarly the load of the already running applications will not violate the QoS requirements of the new application). To calculate the impact of admitting a new stream processing application on the QoS of the existing applications (and likewise, the impact of the running applications on the potential execution of the one to be admitted), a Synergy node that processes a probe utilizes a *QoS Impact Projection algorithm*. This algorithm runs in all nodes with candidate components through which the probes are propagated. The QoS Impact Projection is performed for all the applications that use components on those nodes. The goal is, that, if the projected QoS penalty will cause the new or the existing applications to violate their QoS constraints, these components are not further considered and are thus removed from the candidate list. For example, in Figure 6, candidate components c_{10} and c_{40} are used by existing applications. Assuming that QoS violations would be projected as a result of the new stream workload, c_{10} and c_{40} are not considered as candidate components for the operators o_1 and o_4 respectively, and therefore are grayed out in the Figure. On the contrary, even though c_{20} and c_{39} are used by existing applications, they are still considered as candidate components for the operators o_2 and o_3 respectively, if no QoS violations are projected for them. We now describe the details of the QoS Impact Projection algorithm, first for regular traffic (Section III-D1) and then for bursty traffic (Section III-D2).

1) QoS Impact Projection for Regular Traffic: The QoS Impact Projection algorithm to estimate the effect of component reuse works as follows: For each component c_i, the node estimates its execution time. This includes the processing time τ_{c_i} of the component c_i to execute locally on the node and the queueing time in the scheduler's (FCFS) queue as it waits for other components to complete. The queueing time is defined as the difference between the arrival time of the component invocation at the node and the time the component actually starts executing. We can then determine the mean execution time x_{c_i,v_i} for each component c_i on the node v_i.

For regular traffic, we approximate arrivals of stream data tuples with a Poisson distribution and the durations of their processing with an exponential distribution. Data tuples arrive continuously and the scheduler's queue is large enough to store them until they are processed. Under these assumptions, we can model the application behavior as an M/M/1 system [20]. While such a model can only provide an approximation of the execution time, it is commonly used due to its simplicity and has also been used to represent streaming data [21]. Our experimental results show that this simplified model can provide good projection performance for both synthetic and real datasets. If p_{v_i} represents the load on node v_i hosting component c_i, and τ_{c_i} represents the processing time for c_i to execute isolated on v_i, the mean execution time for component c_i on node v_i is given by $x_{c_i,v_i} = \frac{\tau_{c_i}}{1-p_{v_i}}$. The mean communication time y_{s_j,l_j} on the virtual link l_j for the stream s_j transmitted from component c_i to its downstream component c_j is estimated similarly: It includes the transmission time σ_{s_j} for the stream s_j, and also the queueing delay on the virtual link. If b_{l_j} represents the load (consumed bandwidth) on virtual link l_j connecting component c_i to its downstream component in the application component graph, the mean communication time y_{s_j,l_j} to transmit stream s_j through the virtual link l_j is then given by $y_{s_j,l_j} = \frac{\sigma_{s_j}}{1-b_{l_j}}$. Given the processing times τ_{c_i} and the transmission times σ_{s_j} required respectively for the execution of the components c_i and the data transfer of the streams s_j of an application, as well as the current respective loads p_{v_i} and b_{l_j}, a Synergy node can compute the projected end-to-end execution time for the entire application as $\hat{t} = max_{path} \sum_{v_i \in V_\lambda, l_j \in \lambda} \left(\frac{\tau_{c_i}}{1-p_{v_i}} + \frac{\sigma_{s_j}}{1-b_{l_j}} \right)$, where the max_{path} is used in the cases where the application is represented by a graph with multiple paths, in which case the projected execution time of the entire application is the maximum path delay.

The processing time τ_{c_i} and transmission time σ_{s_j} are derived from the processing and bandwidth requirements, p_{o_i} and b_{s_j} respectively, which are included for the corresponding operators o_i and streams s_j in the query plan ξ. The bandwidth requirements are calculated according to the user-requested stream rate, while the processing requirements are calculated according to the data rate and profiled processing times for the operators [13]. The current processor and network loads, p_{v_i} and b_{l_j} respectively, are known locally at the individual nodes. These values are used to estimate the local impact δ of the component reuse on the existing applications as follows (based on the projected execution time):

Let $\frac{\tau_{c_i}}{1-p_{v_i}}$ denote the mean execution time required for executing component c_i on the node v_i by the application. After sharing the component with the new application, the projected execution time would become: $\frac{\tau_{c_i}}{1-(p_{v_i}+p_{c_i})}$, where $(p_{v_i} + p_{c_i})$ represents the new processing load on the node after reusing the component c_i. p_{c_i} represents the maximum profiled load for c_i. This makes the projection conservative, so that QoS violations will be avoided. Alternatively, a projection can be less pessimistic by using average or minimum instead of maximum load. We then compute the impact δ on the application execution time, as the difference between the projected end-to-end execution time after the reuse, $\hat{t}'$, and the execution time before the reuse, $\hat{t}$:

$$\delta = \hat{t}' - \hat{t} = \frac{\tau_{c_i}}{1-(p_{v_i}+p_{c_i})} - \frac{\tau_{c_i}}{1-p_{v_i}} \tag{6}$$

The projected impact δ is acceptable if $\delta + \hat{t} \leq q_t$, in other words if the new projected execution time is acceptable. In the above inequality, q_t is the requested end-to-end execution time QoS metric that was specified by the user in Q_ξ. Similar to ξ, it is cached for every application on each node that is part of the application. $\hat{t}$ is the current end-to-end execution time for the entire application. $\hat{t}$ is measured by the receiver of a stream processing session and communicated to all nodes participating in it using a feedback loop [16]. This enables the processing to adapt to significant changes in the resource utilization, such as finished applications or execution of new components. For an application that is still in the admission process, $\hat{t}$ is approximated by the sum of the processing and transmission times up to this node, as carried by the probe.

2) QoS Impact Projection for Bursty Traffic: Oftentimes streaming data, such as voice-over-IP data, network traffic, or

sensor measurements generated in an emergency application, can be bursty, and therefore well approximated by an ON/OFF model [11], [22]. In an ON/OFF model, segments of data arrivals with high rate are followed by segments of data arrivals with low rate. Similar to [11], [22], we approximate bursty traffic using an ON/OFF model. Each segment of the ON/OFF traffic represents regular traffic, the arrival of data tuples of which is approximated using a Poisson distribution. We apply an M/M/1 queueing model *within each segment* of the ON/OFF traffic. Thus, using an M/M/1 system we model the traffic within each segment as having constant mean arrival rate of data tuples. Modeling it using M/M/1 allows us to apply queueing theory to estimate the mean execution time within each segment, as was described in Section III-D1. We do not use the same M/M/1 model to generate bursty traffic. Traffic within each segment, which is regular, is approximated using a separate M/M/1 model. A change in the measured mean arrival rate of data tuples signifies the transition to a new segment.

We define a stream segment, denoted by z_i, as a time interval with approximately constant mean arrival rate of data tuples. We partition bursty traffic into a sequence of such stream segments. This way we approximate bursty streaming data by assuming Poisson arrivals of stream data tuples within each segment, but with different rates in different segments. However, the challenge is to identify the correlation between the segments of different streams, because the segments of high and low rates for different streams do not necessarily coincide. To address this challenge we employ the concept of stream signatures [23]. For each stream we construct and maintain a data arrival time series called the signature of the stream, to describe its workload pattern. The signature Ω_j of a stream s_j is a time series of the load associated with processing the data tuples of the stream within a sliding window of length W, $\Omega_j = \{p_1, \ldots p_i, \ldots p_W\}$, where p_i denotes the average processing load for segment z_i in the bursty stream. Measurements are added to the signature of a stream every time the mean arrival rate of data tuples changes, and substitute old measurements after the window is filled. Signatures are stored as arrays of measurements. Each new measurement added to a stream's signature is calculated from the number of data tuples that have arrived since the last measurement, multiplied by the processing load (i.e., percentage of CPU cycles) spent for each data tuple. The signatures of the streams currently being processed by a node are maintained by its monitoring module. For the streams of the application that is currently being admitted, if their signatures are not provided, we either obtain them through off-line profiling, or approximate them using the load measurements of the existing components the application will be using. As the execution of the new application begins, the sliding windows of the signatures of its streams are filled with the actual processing loads.

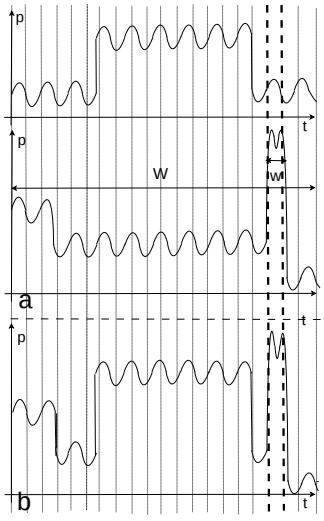

Fig. 8. The two stream signatures shown in a) are aggregated to get the combined signature of b).

As Figure 8 shows, the processing load of sharing a component is calculated as the combination of the processing loads of all of the component's input streams. During a segment z_i, in which the mean arrival rate of data tuples remains constant, the execution time for processing data tuples is approximated by an M/M/1 queueing model. When estimating the workload of multiple input streams, we use the shortest segment length w among the segment lengths of all input streams, as is shown in Figure 8. The benefit of employing stream signatures is two-fold: First, they enable us to identify the shortest segment length w, i.e., the shortest time interval with constant mean arrival rate of data tuples among multiple bursty streams. Second, they enable us to combine the processing loads of multiple bursty streams, by aggregating the measurements of all streams for each segment of minimum length w. Hence, in the example of Figure 8, the shortest segment length w within the sliding window of length W is identified. It is then used to divide the signatures into segments of minimum length and perform the aggregations of the streams' processing loads.

Combining the processing loads of multiple streams by aggregation is possible because of two reasons. The first reason is that we aggregate the processing loads of the individual streams within the shortest segment length w, for which all streams have constant mean arrival rate of data tuples. The second reason is that, assuming an M/M/1 queueing model for each individual stream in each segment of length w, the combination of all streams within that segment also follows an M/M/1 queueing model [20]. Thus, assuming $p_{\Omega_{j,w}}$ represents the measurement (i.e., processing load) belonging to a signature $\Omega_{j,w}$ of a stream s_j for the shortest segment length w, the mean execution time for component c_i on node v_i processing all input streams S_{v_i} in a segment of length w is given by $x_{c_i,v_i,w} = \frac{\tau_{c_i}}{1-\sum_{\Omega_{j,w}\forall s_j \in S_{v_i}} p_{\Omega_{j,w}}}$. After sharing the component with the new application, which incurs additional maximum processing load p_{c_i}, the projected execution time for each segment of length w would become: $\frac{\tau_{c_i}}{1-(\sum_{\Omega_{j,w}\forall s_j \in S_{v_i}} p_{\Omega_{j,w}} + p_{c_i})}$. We can then compute the impact δ_w on the projected execution time for the entire application, for every segment of length w within the window of length W. As alternative admission criteria we can use average, minimum, or maximum projected execution times over all segments to project the impact. δ_w is computed as the difference of the projected end-to-end execution time after the reuse, $\hat{t}'$, from the one before, $\hat{t}$:

$$\delta_w = \hat{t}' - \hat{t} = \frac{\tau_{c_i}}{1-(\sum_{\Omega_{j,w}\forall s_j \in S_{v_i}} p_{\Omega_{j,w}} + p_{c_i})} - \frac{\tau_{c_i}}{1-\sum_{\Omega_{j,w}\forall s_j \in S_{v_i}} p_{\Omega_{j,w}}} \quad (7)$$

The projected impact is acceptable if $\delta_w + \hat{t} \leq q_t, \forall w \in W$, i.e., if the new projected execution time is acceptable for every segment of length w within the window of length W.

Equations (6) and (7) are the formulas used in the QoS Impact Projection algorithm, for regular and for bursty arrival rates respectively. A Synergy node has available, locally, all the required information to compute the impact δ for all applications it is currently participating in. This information

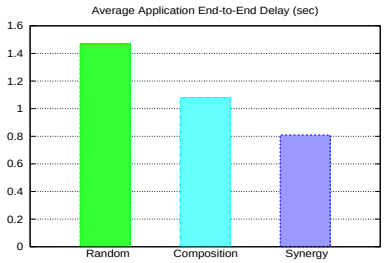

Fig. 9. Average application end-to-end delay.

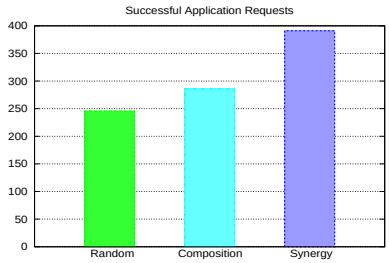

Fig. 10. Successful application requests.

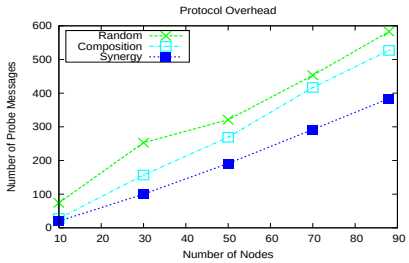

Fig. 11. Protocol overhead.

is available by maintaining local load information, monitoring the local processor utilization, and caching ξ and Q_ξ for all running applications, along with their current end-to-end execution times. Synergy uses the projected application execution time to estimate the effect of the component reuse on the existing applications, by considering the effect of increased processor load on the time required to invoke the component.

This projection is performed for all applications currently invoking a component to be reused, for all applications invoking other components on the node, and also for the application that is to be admitted. If the projected impact is acceptable for all applications, the component can be reused, and the node sends an acknowledgement message to inform its upstream node accordingly. Otherwise, and if there are no other local components that can be reused, the probe is dropped.

IV. Experimental Evaluation

A. Prototype over PlanetLab

1) Methodology: Our Synergy prototype was implemented as a multi-threaded system including about 20000 lines of Java code, running on each of 88 physical nodes of PlanetLab [12]. The implementation was based on the SpiderNet service composition framework [14]. 100 components were deployed uniformly across the nodes, with a replication degree of 5. We used a probing ratio of 10%. Application requests asked for 2 to 4 components chosen randomly and for the corresponding streams between the components. We generated approximately 9 requests per second throughout the system, using a Zipf distribution with $\alpha = 1.6$, expecting stream processing applications to follow trends similar to media streaming and web content provisioning applications [24]. We also experimented with different request distributions in the simulations. We compared Synergy against two different composition algorithms: A Random algorithm that randomly selected one of the candidates for each application component. A Composition algorithm (such as [14]), that performs QoS-aware composition but does not consider result stream reuse or the effects of component reuse on the applications' QoS.

2) Results and Analysis:

Average Application End-to-End Delay. Figure 9 shows the average application end-to-end delay achieved by the three composition approaches for each transmitted data tuple. Synergy offers a 45% improvement over Random and a 25% improvement over Composition. The average end-to-end delay is in the acceptable range of less than a second. Reusing existing result streams offers Synergy an advantage, since the end-to-end delay is reduced for some requests by avoiding redundant stream processing.

Successful Application Requests. An important efficiency metric of a component composition algorithm is the number of requests it manages to accommodate and meet their QoS demands, shown in Figure 10. Synergy successfully accommodates 27% more applications than Composition and 37% more than Random. Random does not take the QoS requirements into account, thus misassigns a lot of requests. While Composition takes operator, resource, and QoS requirements into account, it does not employ QoS impact projection to prevent QoS violations on currently running applications. This results in applications that fail to meet their QoS demands during their execution, due to dynamic arrivals of new requests in the system. In contrast, Synergy manages to increase the capacity of the system and also limit the QoS violations.

Protocol Overhead. We show the overhead of the composition protocols which is attributed to the probe messages in Figure 11. To discover components and streams, we use the DHT-based routing scheme of Pastry, which keeps the number of discovery messages low, while the number of messages needed to probe alternative component graphs quantifies our protocol's overhead. Synergy's sharing-aware component composition manages to reduce the number of probes: By being able to discover and reuse existing streams to satisfy parts or the entire query plan, it keeps the number of candidate components that need to be probed smaller. Also important is that the overhead grows linearly with the number of nodes in the system, which allows the protocol to scale to larger numbers of nodes. The probing ratio is another knob that can be used to tune the protocol overhead further [5]. While Random's overhead could also be tuned to allow less candidates to be visited, its per hop selections would still be QoS-unaware.

Average Setup Time. Table 15 shows the breakdown of the average time needed for an application setup, for the three composition algorithms. The setup time is divided into time spent to discover components and streams, and time spent to probe candidate components. As is shown, the discovery of streams and components is only a small part of the time needed to set up a stream processing session. Most of the time is spent in transmitting probes to candidate components and running

Setup Time (ms)	Random	Composition	Synergy
Discovery	240	188	243
Probing	4509	4810	3141
Total	4749	4998	3384

Fig. 15. Breakdown of average setup time.

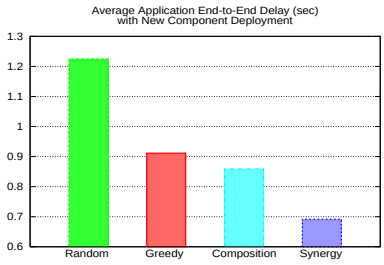

Fig. 12. Average end-to-end delay with deployment.

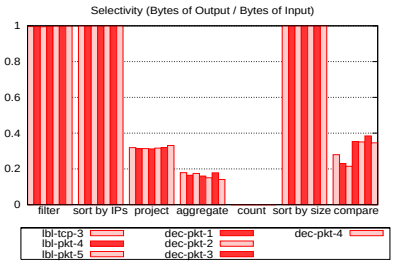

Fig. 13. Selectivity of different operators.

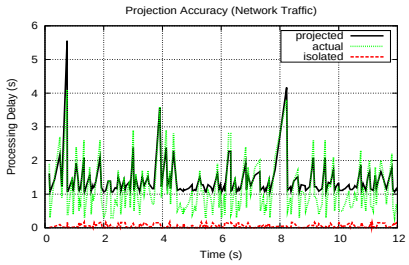

Fig. 14. Projection accuracy for network traffic.

the composition algorithm. Sharing streams allows Synergy to save time from component probing, which effectively leads to 32% faster setup time than Composition. The total setup time is only a few seconds. Having to discover less components balances out the cost of having to discover streams. Discovering a stream, especially if it is the final output of the query plan, can render multiple component discoveries unnecessary.

B. Simulations

1) Methodology: To further evaluate the performance of Synergy's sharing-aware composition algorithm, we implemented a distributed stream processing simulator in about 8500 lines of C++ code. The network topology fed to the simulator was a transit-stub topology of 1500 routers, generated by the GT-ITM Internet topology generator [25]. We simulated a large overlay network of 500 nodes chosen randomly from the underlying topology. Nodes and links were assigned processing and communication capacities from discrete classes, to simulate a heterogeneous system. 1000 components were distributed uniformly across the nodes of the system, with a uniform replication degree of 5; i.e., 200 unique components and 800 component replicas were deployed at the nodes. Application requests (i.e., query plans) consisted of requests with 2 to 10 operators chosen randomly. For each application, we set its QoS requirement 30% higher than the time needed for the application to execute in isolation. We investigated both the performance of Synergy's composition algorithm and its sensitivity to the above parameters and those results are presented in [26]. We compared Synergy not only against Random and Composition, but also against a Greedy algorithm that at each composition step selected the candidate component that resulted in the minimum next-hop delay. Note that this does not necessarily result in the minimum end-to-end delay for the entire application. To implement this algorithm in a distributed prototype some delay monitoring service such as the ones discussed in [17] would be needed.

2) Results and Analysis: In this set of experiments we investigated the performance of Synergy's collocation-based component deployment strategy, described in Section III-B.

Average Application End-to-End Delay. To trigger new component deployment, we included in the query plan of each application request an operator that was not offered by any component in the system. We kept query plan sizes uniformly distributed from 2 to 10 operators, as mentioned in Section IV-B1. Synergy collocated the new component with another component of the application component graph, based on the heuristics described in Section III-B, also performing the required resource and QoS checks. Composition and Greedy deployed the new component on the node that had the minimum delay from the upstream node, i.e., from the node hosting the previous component in the application component graph. Additionally, Composition selected the next closest node if the deployment would cause a resource violation. Finally, Random blindly selected a node to deploy the new component. Figure 12 shows the average application end-to-end delay achieved by the different algorithms. The execution delay is averaged over 100 application instantiations. Synergy's collocation-based component deployment reduces average end-to-end delay by approximately 20% over the delay-based deployment of Composition and Greedy. Furthermore, it does not require maintaining delay information. Hence, it is an attractive strategy for infrequent component deployment. When many components need to be deployed, in which case resource and QoS violations due to the collocation of multiple components may be more frequent, techniques for placing a complete component graph may be considered [7], [18].

Selectivity. Synergy's collocation-based deployment takes the operators selectivity into account to minimize network traffic across components. We investigated the selectivity of operators of a real stream processing application operating on real streams, to quantify the traffic reduction. We implemented a top-k network traffic monitoring application (http://synergy.cs.ucr.edu/screenshots.html) from the stream query repository [27] and recorded the output of the operators for streams produced by seven different traces of network traffic from the Internet traffic archive [28]. Figure 13 shows that for three out of seven operators of the query plan, average traffic reduction reaches 69%, 82%, and 72% respectively, while for the count operator traffic is reduced to just one data tuple. While the traffic reduction depends on the operator semantics, it is consistent among different datasets, making selectivity an important factor when deploying components.

In the next set of experiments we examined the accuracy of Synergy's QoS impact projection algorithm described in Section III-D. In particular we looked at how the projected processing delay of individual streams compared to the actual processing delay experienced by the data tuples of these streams, by experimenting with both real and synthetic datasets. In all figures we also show the processing delay of the isolated stream, that is, the processing delay if no queueing for processing other streams existed.

Projection accuracy for real network traffic. We investigated

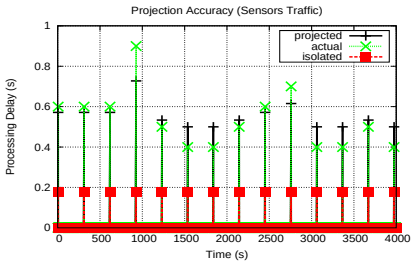

Fig. 16. Projection accuracy for sensor traffic.

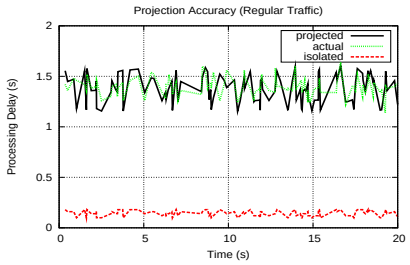

Fig. 17. Projection accuracy for regular traffic.

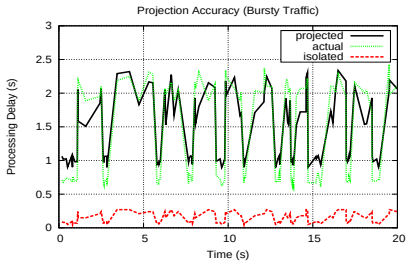

Fig. 18. Projection accuracy for bursty traffic.

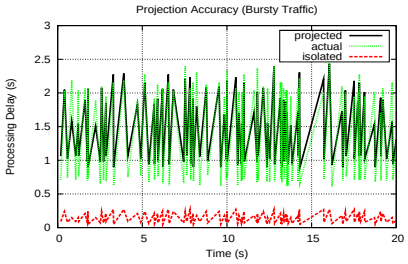

Fig. 19. Projection accuracy for bursts with period 0.5 second.

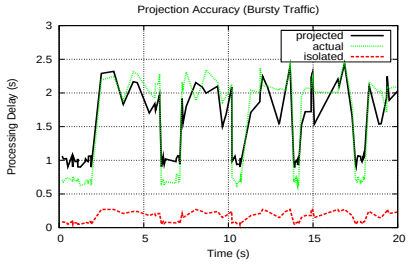

Fig. 20. Projection accuracy for bursts with period 5 seconds.

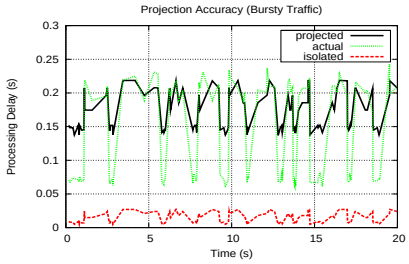

Fig. 21. Projection accuracy for bursts with high rate 3 tuples/s and low rate 1 tuple/s.

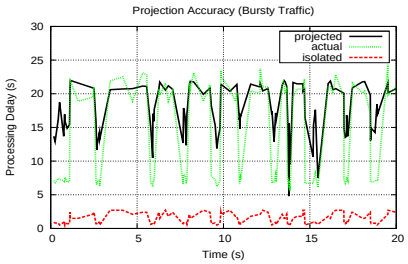

Fig. 22. Projection accuracy for bursts with high rate 300 tuples/s and low rate 100 tuples/s.

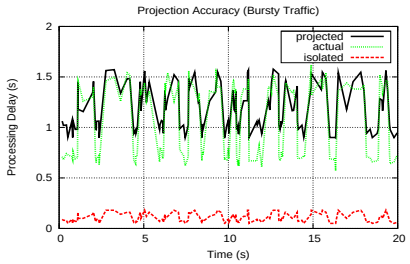

Fig. 23. Projection accuracy for bursts with high rate 20 tuples/s and low rate 10 tuples/s.

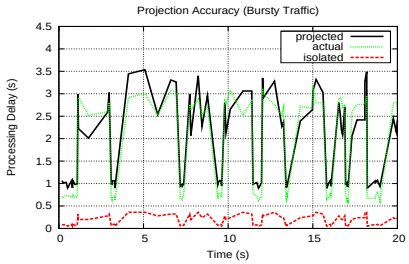

Fig. 24. Projection accuracy for bursts with high rate 40 tuples/s and low rate 10 tuples/s.

the projection accuracy for processing a trace of TCP traffic between the Lawrence Berkeley Laboratory and the rest of the world, which was trace LBL-TCP-3 from the Internet traffic archive [28]. Each data tuple was 192 bits long, and contained a timestamp, and fields defining the source, destination, and size of packets exchanged. As can be seen in Figure 14, the generated stream was bursty and did not follow any easily identifiable pattern. Synergy's QoS impact projection follows the bursts very closely, projecting processing delays close to the ones experienced. The projections for the low rate segments are mostly above the actual delays, which may lead to more conservative compositions, but no QoS violations.

Projection accuracy for real sensor traffic. Next we investigated the projection accuracy for bursty streams that followed a pattern, specifically the data streams produced by sensors installed in redwood trees collected by the UC Berkeley Sonoma dust project [29]. Each data tuple produced was 352 bits long, and contained a timestamp, multiple fields characterizing the sensor that produced it, as well as a variety of measurements, including humidity. A burst of measurements lasting approximately one second was generated every five minutes. Figure 16 shows that these periodic bursts were followed closely by Synergy's projection algorithm, which accurately identified the segments of high and zero rate.

Projection accuracy for synthetic regular traffic. We next generated regular traffic, with data tuples arriving at a rate of 20 tuples/second and following a Poisson distribution. Figure 17 shows that the processing delay trends are followed closely by Synergy's projection algorithm, while the projected delay values are in a close range to the actual ones.

Projection accuracy for synthetic bursty traffic. We also generated bursty traffic with a period of 2.5 seconds, high rate of 30 tuples/second and low rate of 10 tuples/second. Figure 18 shows that Synergy's projection algorithm accurately identifies the high and low rate segments. Similar to the projection for network traffic of Figure 14, the projections for the low rate segments are mostly conservative, i.e., above the actual delays. However, most importantly, the high rate segment projections are not optimistic, and therefore do not lead to QoS violations.

Finally, we investigated the accuracy of Synergy's QoS impact projection under various conditions, by changing individual parameters of the synthetic bursty streams, while keeping the rest of them as in the experiment of Figure 18.

Sensitivity to burst period. Figures 19 and 20 show the projection accuracy for bursty traffic with periods of 0.5 and 5 seconds respectively. As can be seen, the length of the bursts

does not affect the accuracy with which the algorithm identifies segments of low and high rate.

Sensitivity to burst rate. Figures 21 and 22 show the projection accuracy when varying the burst rate. Figure 21 shows traffic with high rate of 3 tuples/second and low rate of 1 tuple/second, while Figure 22 shows traffic with high rate of 300 tuples/second and low rate of 100 tuples/second. We observe that these variations in rate make more evident the conservative projection for low rate segments described in Figure 18, which may lead to more conservative compositions, but not to QoS violations. We note that we have not observed such extreme rates for either of the two real traffic datasets.

Sensitivity to burst ratio. Figures 23 and 24 show the projection accuracy when varying the ratio of the rates of the high- and low-rate segments. Figure 23 shows traffic with high rate of 20 tuples/second and low rate of 10 tuples/second, while Figure 24 shows traffic with high rate of 40 tuples/second and low rate of 10 tuples/second. We observe that the ratio of the high and low rates does not affect the detection of segments, nor the accuracy with which processing delays are projected.

V. Related Work

Distributed stream processing [3], [8] has been the focus of several recent research efforts from many different perspectives. In [7], [18] the placement problem of a complete component graph in a DSPS to make efficient use of the network resources and maximize query performance is discussed. Our work is complementary, in that our focus is on the effects of sharing existing components, and we address partial component graph deployment only when previously deployed components cannot be reused. While [7] mentions component reuse, they do not focus on the impact on already running applications. [6] describes an architecture for distributed stream management that makes use of in-network data aggregation to distribute processing and reduce communication overhead. A clustered architecture is assumed, as opposed to Synergy's totally decentralized protocols. Service partitioning to achieve load balancing taking into account the heterogeneity of the nodes is discussed in [30]. While a balanced load is the final selection criterion among candidate component graphs in Synergy as well, our focus is on QoS provisioning. The distributed composition probing approach is first presented in [5], [14]. Synergy extends this work by considering stream reuse and evaluating the impact of component sharing. Our techniques for distributed stream processing composition apply directly to multimedia streams [16] as well. This paper builds upon our earlier work [26] by augmenting the composition protocol with new component deployment, extending the QoS impact projection algorithm to handle both regular and bursty streams, and experimentally investigating the projection accuracy.

While we focus on component composition for stream processing, our techniques may apply to other composite applications with QoS requirements, such as QoS-sensitive web services. Similar to Synergy, works on web service composition [31]–[33] take into account QoS metrics. They discuss dynamic composition algorithms that select web services so that utility is maximized [31], end-to-end QoS is guaranteed [32], or maximized [33]. [32] proposes heuristics with near-optimal solutions in polynomial time, while [33] presents optimal solutions using integer programming. The main difference between these approaches and Synergy is that they propose centralized solutions that rely on global knowledge, whereas Synergy employs a distributed composition protocol. While centralized solutions might be appropriate for a web services environment, a distributed approach is more suitable for highly-dynamic and very large-scale distributed stream processing environments [5]–[9]. However, our maximum sharing discovery and QoS impact projection algorithms are independent of the composition protocol and may be combined with existing web service composition approaches [31]–[33].

VI. Conclusions and Future Work

In this paper we have presented Synergy, a distributed stream processing middleware that provides sharing-aware component composition. Synergy is built on top of a totally decentralized overlay architecture and utilizes a Maximum Sharing Discovery algorithm to reuse existing streams, and a QoS Impact Projection algorithm to reuse existing components and yet ensure that the QoS requirements of the currently running applications will not be violated. Both our prototype implementation of Synergy over PlanetLab and our simulations of its composition algorithm show that sharing-aware component composition can enhance QoS provisioning for distributed stream processing applications.

Our future work includes the integration of iterative execution of Synergy's composition protocol with techniques for increasing application reliability. This can enable DSPSs that are more reliable in the presence of overloads, as well as more robust against node or link failures at run-time or during composition. Proactive migration [34] can guard against QoS violations. On the other hand, predictive failure management [35], and availability-aware placement [36] can protect against component or node failures. To offer fault tolerance, either reactive or proactive failure recovery schemes can be used [10]. In reactive recovery a new application component graph is composed upon failure, while in proactive recovery backup application component graphs are maintained. Integrating replication with composition can also increase fault tolerance. To that end, consistency trade-offs [37], and checkpoint scheduling [38] are worth investigating.

References

[1] S. Chandrasekaran *et al.*, "TelegraphCQ: Continuous dataflow processing for an uncertain world," in *Proceedings of CIDR, Asilomar, CA, USA*, January 2003.

[2] R. Motwani *et al.*, "Query processing, resource management, and approximation in a data stream management system," in *Proceedings of CIDR, Asilomar, CA, USA*, January 2003.

[3] D. Abadi *et al.*, "The design of the Borealis stream processing engine," in *Proceedings of CIDR, Asilomar, CA, USA*, January 2005.

[4] L. Chen, K. Reddy, and G. Agrawal, "GATES: A grid-based middleware for distributed processing of data streams," in *Proceedings of IEEE HPDC-13, Honolulu, HI, USA*, June 2004.

[5] X. Gu, P. Yu, and K. Nahrstedt, "Optimal component composition for scalable stream processing," in *Proceedings of 25th ICDCS, Columbus, OH, USA*, June 2005.

[6] V. Kumar, B. Cooper, Z. Cai, G. Eisenhauer, and K. Schwan, "Resource-aware distributed stream management using dynamic overlays," in *Proceedings of 25th ICDCS, Columbus, OH, USA*, June 2005.

[7] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer, "Network-aware operator placement for stream-processing systems," in *Proceedings of 22nd ICDE, Atlanta, GA, USA*, April 2006.

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

[9] K. L. Wu *et al.*, "Challenges and experience in prototyping a multi-modal stream analytic and monitoring application on System S," in *Proceedings of 33rd VLDB, Vienna, Austria*, September 2007.

[10] X. Gu and K. Nahrstedt, "On composing stream applications in peer-to-peer environments," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 8, pp. 824–837, July 2006.

[11] M. Hammad, M. Franklin, W. Aref, and A. Elmagarmid, "Scheduling for shared window joins over data streams," in *Proceedings of 29th VLDB, Berlin, Germany*, September 2003.

[12] A. Bavier *et al.*, "Operating system support for planetary-scale network services," in *Proceedings of NSDI, San Francisco, CA*, March 2004.

[13] T. Abdelzaher, "An automated profiling subsystem for QoS-aware services," in *Proceedings of 6th IEEE RTAS, Washington, DC*, June 2000.

[14] X. Gu and K. Nahrstedt, "Distributed multimedia service composition with statistical QoS assurances," *IEEE Transactions on Multimedia*, vol. 8, no. 1, pp. 141–151, February 2006.

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

[16] F. Chen, T. Repantis, and V. Kalogeraki, "Coordinated media streaming and transcoding in peer-to-peer systems," in *Proceedings of 19th IPDPS, Denver, CO, USA*, April 2005.

[17] J. Ledlie, P. Gardner, and M. Seltzer, "Network coordinates in the wild," in *Proceedings of NSDI, Cambridge, MA, USA*, April 2007.

[18] Y. Ahmad and U. Çetintemel, "Network-aware query processing for stream-based applications," in *Proceedings of 30th VLDB, Toronto, Canada*, August 2004.

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

[20] L. Kleinrock, *Queueing Systems. Volume 1: Theory*. New York, NY, USA: John Wiley and Sons Inc., 1975.

[21] N. Antunes, C. Fricker, F. Guillemin, and P. Robert, "Integration of streaming services and TCP data transmission in the Internet," *Elsevier Performance Evaluation*, vol. 62, no. 1–4, pp. 263–277, October 2005.

[22] A. Markopoulou, F. Tobagi, and M. Karam, "Assessing the quality of voice communications over Internet backbones," *IEEE/ACM Transactions on Networking*, vol. 11, no. 5, pp. 747–760, October 2003.

[23] X. Gu, Z. Wen, and P. Yu, "BridgeNet: An adaptive multi-source stream dissemination service overlay," in *Proceedings of IEEE INFOCOM, Anchorage, AK, USA*, May 2007.

[24] L. Cherkasova and M. Gupta, "Analysis of enterprise media server workloads: Access patterns, locality, content evolution, and rates of change," *IEEE/ACM Transactions on Networking*, vol. 12, no. 5, pp. 781–794, October 2004.

[25] E. Zegura, K. Calvert, and S. Bhattacharjee, "How to model an internetwork," in *Proceedings of IEEE INFOCOM, San Francisco, CA, USA*, March 1996.

[26] T. Repantis, X. Gu, and V. Kalogeraki, "Synergy: Sharing-aware component composition for distributed stream processing systems," in *Proceedings of 7th ACM/IFIP/USENIX Middleware*, November 2006.

[27] Stream Query Repository: Network Traffic Management, "http://infolab.stanford.edu/stream/sqr/netmon.html."

[28] The Internet Traffic Archive, "http://ita.ee.lbl.gov/html/traces.html."

[29] UC Berkeley Sonoma Dust, "http://www.cs.berkeley.edu/~get/sonoma/data.ht

[30] B. Gedik and L. Liu, "PeerCQ: A decentralized and self-configuring peer-to-peer information monitoring system," in *Proceedings of 23rd ICDCS, Providence, RI, USA*, May 2003.

[31] D. Menasce, "Composing web services: A QoS view," *IEEE Internet Computing*, vol. 8, no. 6, pp. 88–90, November/December 2004.

[32] T. Yu, Y. Zhang, and K. Lin, "Efficient algorithms for web services selection with end-to-end QoS constraints," *ACM Transactions on the Web*, vol. 1, no. 1, pp. 1–26, May 2007.

[33] L. Zeng, B. Benatallah, A. Ngu, M. Dumas, J. Kalagnanam, and H. Chang, "QoS-aware middleware for web services composition," *IEEE Transactions on Software Engineering*, vol. 30, no. 5, pp. 311–327, May 2004.

[34] T. Repantis and V. Kalogeraki, "Hot-spot prediction and alleviation in distributed stream processing applications," in *Proceedings of 38th DSN, Anchorage, AK, USA*, June 2008.

[35] X. Gu, S. Papadimitriou, P. S. Yu, and S. P. Chang, "Toward predictive failure management for distributed stream processing systems," in *Proceedings of 28th ICDCS, Beijing, China*, June 2008.

[36] T. Repantis and V. Kalogeraki, "Replica placement for high availability in distributed stream processing systems," in *Proceedings of 2nd DEBS, Rome, Italy*, July 2008.

[37] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker, "Fault-tolerance in the Borealis distributed stream processing system," in *Proceedings of ACM SIGMOD, Baltimore, MD, USA*, June 2005.

[38] J. Hwang, Y. Xing, U. Çetintemel, and S. Zdonik, "A cooperative, self-configuring high-availability solution for stream processing," in *Proceedings of 23rd ICDE, Istanbul, Turkey*, April 2007.

Thomas Repantis is a PhD candidate at the Computer Science and Engineering Department of the University of California, Riverside. His research interests lie in the area of distributed systems, distributed stream processing systems, middleware, peer-to-peer systems, pervasive and cluster computing. He holds an MSc from the University of California, Riverside and a Diploma from the University of Patras, Greece, and has interned with IBM Research, Intel Research and Hewlett-Packard.

Xiaohui Gu is an assistant professor in the Department of Computer Science at North Carolina State University. She was a research staff member at IBM T. J. Watson Research Center, Hawthorne, New York, between 2004 and 2007. Her general research interests include distributed systems, operating systems, and computer networks with a current focus on autonomic system management using machine learning methods, massive data stream processing, peer-to-peer systems, and mobile systems. She received ILLIAC fellowship, David J. Kuck Best Master Thesis Award, and Saburo Muroga Fellowship from University of Illinois at Urbana-Champaign. Dr. Gu is a recipient of IBM Faculty Award 2008. She received her PhD degree in 2004 and MS degree in 2001 from the Department of Computer Science, University of Illinois at Urbana-Champaign. She received her BS degree in computer science from Peking University, Beijing, China.

Vana Kalogeraki is an associate professor at the University of California, Riverside. Her research interests include distributed and real-time systems, peer-to-peer systems and distributed sensor systems. She received her Ph.D. from the University of California, Santa Barbara in 2000. In 2001-2002, she held a Research Scientist position at Hewlett-Packard Labs in Palo Alto, CA. She has published numerous technical papers, including co-authoring the Object Management Group (OMG) CORBA Dynamic Scheduling Standard. She has delivered tutorials and seminars on peer-to-peer computing. She has organized and served on program committees for several technical conferences. She has served as the General Chair of the "14th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS 2006)", Program co-Chair of the "10th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2007)", the "IEEE International Conference on Pervasive Services (ICPS 2005)", the "13th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS 2005) and the "International Workshop on Databases, Information Systems and Peer-to-Peer Computing" at VLDB 2003. She is currently an Associate Editor for the Ad hoc Networks Journal and the Computer Standards & Interfaces Journal. Her research is supported by NSF.