Synergy: Sharing-Aware Component Composition for Distributed Stream Processing Systems

Thomas Repantis[1], Xiaohui Gu[2], and Vana Kalogeraki[1]⋆

[1] Dept. of Computer Science & Engineering, University of California, Riverside, CA 92521
{trep,vana}@cs.ucr.edu
[2] IBM T.J. Watson Research Center, Hawthorne, NY 10532
{xiaohui}@us.ibm.com

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 distributed stream processing middleware that provides sharing-aware component composition. Synergy enables efficient reuse of both data streams and processing components, while composing distributed stream processing applications with QoS demands. Synergy provides a set of fully distributed algorithms to discover and evaluate the reusability of available data streams and processing components when instantiating new stream applications. For QoS provision, Synergy performs QoS impact projection to examine whether the shared processing can cause QoS violations on currently running applications. 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 provision than previously proposed schemes, by judiciously sharing streams and processing components during application composition.

Keywords: Distributed Stream Processing, Component Composition, Shared Processing, Quality-of-Service, Resource Management.

1 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]. In a typical stream processing application, stream processing *components* process continuous data streams in real-time [3] to generate outputs of interest or to identify meaningful events. Often, the data sources, as well as the components that implement the application logic

⋆ The work of the third author is supported by NSF Award 0330481.

are distributed across multiple sites, constituting distributed stream processing systems (DSPSs) (e.g., [4–9]). Stream sources often produce large volumes of data in 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 stream processing middleware design.

While a DSPS provides the components that are needed for an application execution, a major challenge still remains: Namely, how to select among different component instances to compose stream processing applications on-demand. While previous efforts have investigated several aspects of component composition [6, 7] and placement [8] 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 i) previously generated streams and ii) already deployed stream processing components. The distinct characteristics of distributed stream processing applications make sharing-aware component composition particularly challenging. First, stream processing applications often have minimum quality-of-service (QoS) requirements (e.g., end-to-end service 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, the component composition must be performed quickly, during runtime, and be able to adapt to dynamic stream environments. Third, a DSPS needs to scale to a large number of streams and components, which makes centralized approaches inappropriate, since the global state of a large-scale DSPS is changing much faster than it can be communicated to a single host. Hence, a single host cannot make accurate global decisions.

Despite the aforementioned challenges, there are significant benefits to be gained from a flexible sharing-aware component composition: i) *enhanced QoS provision* (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. As a result, the overall system's processing capacity is maximized to meet the scalability requirements of serving many concurrent application requests.

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[1] 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:

[1] In this paper, we focus on the end-to-end execution time QoS metric, consisting of both processing delays at different components and network delays between components.

– We propose a decentralized light-weight composition algorithm that can discover streams and components at run-time and check 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 such that the application resource requirements are met and the workloads at different hosts are balanced.
– 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 the 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 the QoS provision for both current applications and the new application admitted in the system.
– We have implemented a prototype of Synergy and evaluated its performance on the PlanetLab [10] 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 provision 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) Synergy's decentralized composition protocol has low message overhead and offers minimal setup time, in the order of a few seconds.

The rest of the paper is organized as follows: Section 2 introduces the system model. Section 3 discusses Synergy's decentralized sharing-aware component composition approach and its QoS impact projection algorithm. Section 4 presents an extensive experimental evaluation of our system. Section 5 discusses related work. Finally, the paper concludes in Section 6.

2 System Model

In this section, we present the stream processing application model, describe the architecture of the Synergy middleware and provide an overview of its operation. Table 1 summarizes the notations we use while discussing our model.

2.1 Stream Processing Application Model

A data stream s_i 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 $\sum is_i$ and produces a set of output streams $\sum os_i$. Stream processing components can have more than one inputs

Notation	Meaning	Notation	Meaning
c_i	Component	l_i	Virtual Link
o_i	Operator	s_i	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_i}	Network Load on Virtual Link l_i
rp_{v_i}	Residual Processing Capacity on Node v_i	rb_{l_i}	Residual Network Bandwidth on Virtual Link l_i
τ_{c_i}	Processing Time for c_i	x_{c_i,v_i}	Mean Execution Time for c_i on v_i
σ_{s_i}	Transmission Time for s_i	y_{s_i,l_i}	Mean Communication Time for s_i on l_i
q_t	Requested End-to-End Execution Time	$\hat{t}$	Projected End-to-End Execution Time
p_{o_i}	Processing Time Required for o_i	b_{s_i}	Bandwidth Required for s_i

Table 1. Notations.

(*e.g.* a join operator) and outputs (*e.g.* a split operator). Each atomic operator can be provided by multiple component instances $c_1, \ldots, c_k$. 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 [11] (e.g., o_i.name = Aggregator.COUNT, s_i.name = Video.MPEGII.Birthday).

A stream processing request (query) is described by a *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[2]. The CPU 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 bandwidth requirements are calculated according to the user-requested stream rate, while the processing requirements are calculated according to the data rate and resource profiling results for the operators [12]. The stream processing request also specifies the end-to-end QoS requirements $Q_\xi = [q_1, ...q_m]$, such as end-to-end execution time and loss rate. Although our schemes are generic to additive QoS metrics, we focus on the end-to-end execution time metric 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.

The query plan can be dynamically instantiated into different *application component graphs*, denoted by λ, depending on the processing and networking 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 our 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.

2.2 Synergy Architecture

Synergy is a wide-area middleware that consists of a set of distributed hosts v_i connected via virtual links l_i into an overlay mesh on top of the existing IP network.

[2] In general, there may be multiple query plans that can satisfy a stream processing request. Query plan optimization however involves application semantics and is outside the scope of this paper. Thus, in this work we assume the query plan is given.

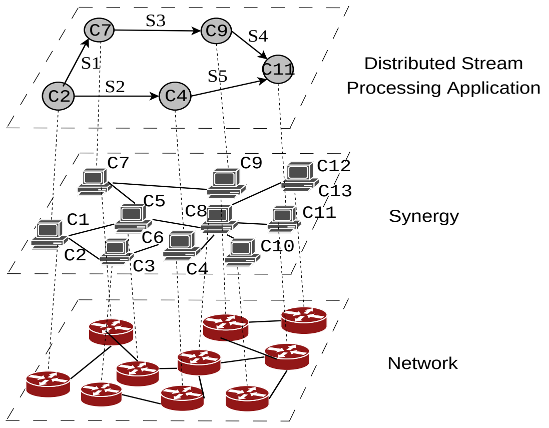


Fig. 1. Synergy system architecture.

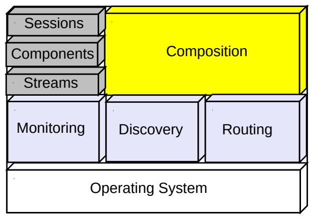


Fig. 2. Synergy node structure.

Synergy as a distributed stream processing middleware undertakes the component composition role to enable stream and component reusability while offering QoS management. Figure 1 shows an overview of our architecture. Synergy leverages the underlying overlay network for registering and discovering available components and streams in a decentralized manner. In our current Synergy prototype we implement a keyword-based discovery service [13] on top of the Pastry distributed hash table (DHT) [14]. However, our middleware can also be integrated with other DHTs, or unstructured overlays [15], since discovery is an independent module of our system. Synergy adopts a fully distributed architecture, where any node of the middleware can compose a distributed stream processing application. After a stream processing request is submitted and a query plan is produced, Synergy is responsible for selecting existing streams that satisfy the query and candidate components that can provide the required operators.

Each Synergy node, denoted by v_i, as illustrated in Figure 2, 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; 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. In the current implementation, the monitoring module can keep 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., [16]). We finally use b_{l_j} to denote the amount of current bandwidth consumed on l_j.

2.3 Approach Overview

We now briefly describe the basic operations of the Synergy middleware. 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. A query plan ξ is produced, that specifies the required operators and the order in which they need to be applied to execute the query. 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.

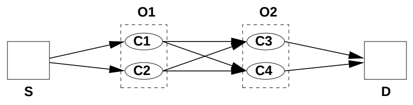


Fig. 3. Probing example.

To avoid redundant computations, the system first tries to discover whether any of the requested streams have been generated by previously instantiated query plans, by querying the overlay infrastructure. To maximize the sharing benefit, the system reuses the result stream(s) generated during the latest possible stages in the query plan. Thus, the system only needs to instantiate the remaining query plan for processing the reusable existing stream(s), to generate the user requested stream(s). The system then probes those candidate nodes that can provide operators needed in the query plan, to determine: i) whether they have the available resources to accommodate the new application, ii) whether the end-to-end latency is within the required QoS, and iii) whether the impact of the new application would cause QoS violations to existing applications. 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. The detailed operation of Synergy's sharing-aware component composition is described in the next section.

3 Design and Algorithm

In this section, we describe the design and algorithm details of our Synergy distributed stream processing middleware, that offers sharing-aware component composi-

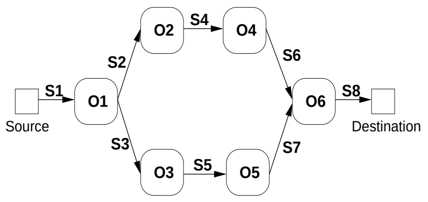


Fig. 4. Query plan example.

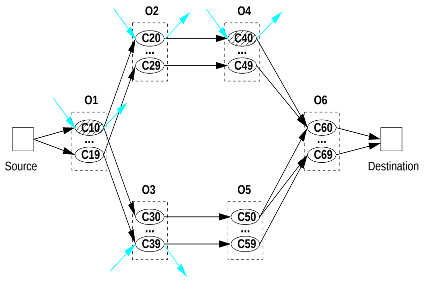


Fig. 5. Synergy composition example.

tion. Synergy can i) reuse existing data streams to avoid redundant computations, and ii) reuse existing components if the new stream load does not lead to QoS violations of the existing applications. We first describe the decentralized component composition protocol, followed by the detailed algorithms for stream reuse and component sharing. Synergy's fully distributed and light-weight composition protocol is executed when instantiating a new application.

3.1 Synergy Composition Protocol

Given a stream processing request, the 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 4 shows an example of a query plan, while Figure 5 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., [6, 13]) 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 1, consists of the following 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 4, starting from the destination, v_s will first check if the result stream (stream s_8) is available. If not, it will look for the streams one hop away from the destination (streams s_6 and s_7), then two hops away from the destination (streams 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

Algorithm 1 Synergy composition.

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

maximum sharable points and about discovering sharable streams and components are described in Section 3.2. 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 [6]. 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 than the one starting after the maximum sharable points.

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: i) 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[3]. ii) 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

[3] In the general case, where other node resources such as memory or disk space are to be taken into account in addition to the processing capacity, congruent equations have to hold for them as well.

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 [12]. iii) 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 3.3. Similarly, the impact of the existing stream processing sessions on the QoS of the new one 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 probing overhead. Otherwise, the node performs *transient* resource allocation to avoid conflicting resource admissions (overallocations) caused by concurrent probes for different requests. The transient resource allocation 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 overlay infrastructure. 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 latency monitoring service [17] is available– the ones with the smallest communication latency are selected. If no candidate components for the next operator are found, a new component has to be deployed. We choose to *collocate* this new component with the current one, deploying it in the same node, if processing resources are available, as this approach minimizes the communication delay between the two components. Other approaches for choosing an appropriate location with regards to future needs can also be employed [8, 18]. Since the probe processing checks will take place for the new component as well, possible resource or QoS violations can be detected. While the resource allocation is transient, the component deployment is permanent. If the particular application session is not established through this path, the newly deployed component might serve other stream processing sessions.

After the candidate components have been selected, v_i *spawns* new probes from P_i for 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.

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 5, 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:

$$operator\ \ constraints : c_i.o = o_i,\ \forall o_i \in \xi, \exists c_i \in \lambda \tag{1}$$
$$QoS\ \ constraints : q_r^\lambda \le q_r^\xi, 1 \le r \le m \tag{2}$$
$$processing\ \ capacity\ \ constraints : rp_{v_i} \ge 0, \forall v_i \in V_\lambda \tag{3}$$
$$bandwidth\ \ constraints : rb_{l_j} \ge 0,\ \ \forall l_j \in \lambda \tag{4}$$

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 the selected composition.

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

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), the system node returns a failure message. If all probes were dropped, apparently the existing components are too overloaded to accommodate the requested application with the specified QoS requirements, or nodes in the probing path are too overloaded to host components that need to be deployed. New components can then be instantiated in strategically chosen places in the network [8, 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 new component deployment, which is outside the scope of this paper. Furthermore, 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.

3.2 Maximum Stream Sharing

Synergy utilizes a peer-to-peer overlay of the nodes in the system for registering and discovering the available components and streams in a decentralized manner. As was mentioned in Section 2.2, the current Synergy implementation is built over Pastry [14]. 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 2.1, each component and stream is given a name, based on a common ontology [11]. 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 duplicated components or streams being stored in the same node. Configuration changes caused by node arrivals and departures are handled gracefully by the DHT. Whenever components are instantiated 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 4 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 system node *backtracks* hop-by-hop the query plan to find whether preceding intermediate result streams exist. For example, in Figure 4, 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 those 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 that might be needed, the peer-to-peer overlay is utilized as was described.

3.3 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 to 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 to the QoS of the existing ones (and also the impact of the running applications to 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. 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 set. For example, in Figure 5, candidate components c_{10} and c_{40} are used by existing applications and with the new stream workload QoS violations are projected. Thus, c_{10} and c_{40} are not considered as candidate components for the operators o_1 and o_4 respectively. 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, because no QoS violation is projected for them.

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 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 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. We assume a simple application behavior approximated by an M/M/1 queueing model for the execution time. Our experimental results show that this simplified model can provide good projection performance. If p_{v_i} represents the load on the node hosting component c_i, the mean execution time for component c_i on node v_i is given by:

$$x_{c_i,v_i} = \frac{\tau_{c_i}}{1 - p_{v_i}} \tag{6}$$

The mean communication time y_{s_i,l_i} on the virtual link l_i for the stream s_i transmitted from component c_i to its downstream component c_j is estimated similarly: It includes the transmission time σ_{s_i} for the stream s_i, and also the queueing delay on the virtual link. If b_{l_i} represents the load (consumed bandwidth) on virtual link l_i connecting component c_i, the mean communication time y_{s_i,l_i} to transmit stream s_i through the virtual link l_i is then given by:

$$y_{s_i,l_i} = \frac{\sigma_{s_i}}{1 - b_{l_i}} \tag{7}$$

Given the processing times τ_{c_i} and the transmission times σ_{s_i} required respectively for the execution of the components c_i and the data transfer of the streams s_i of an application, as well as the current respective loads p_{v_i} and b_{l_i}, a Synergy node can compute the projected end-to-end execution time for the entire application as:

$$\hat{t} = max_{path} \sum_{v_i \in V_\lambda, l_i \in \lambda} \left(\frac{\tau_{c_i}}{1 - p_{v_i}} + \frac{\sigma_{s_i}}{1 - b_{l_i}} \right) \tag{8}$$

where the max_{path} is used in the cases where the application is represented by a graph with more than one paths, in which case the projected execution time of the entire application is the maximum path latency. The processing τ_{c_i} and transmission σ_{s_i} times are however easily extracted from the p_{o_i} and b_{s_i} values which are included for the corresponding operators o_i and streams s_i in the query plan ξ and have been calculated by combining the user requests with profiling [12]. The current loads p_{v_i} and b_{l_i} 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:

Let $\frac{\tau_{c_i}}{1-p_{v_i}}$ denote the mean execution time required for invoking 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: $\frac{\tau_{c_i}}{1-(p_{v_i}+\tau_{c_i})}$, where $(p_{v_i}+\tau_{c_i})$ represents the new processing load on the node after reusing the component. We can then compute the impact δ in the projected execution time for the entire application, as the difference of the projected end-to-end execution time after the reuse, $\hat{t}'$, from the one before the reuse, $\hat{t}$:

$$\delta = \hat{t}' - \hat{t} = \frac{\tau_{c_i}}{1-(p_{v_i}+\tau_{c_i})} - \frac{\tau_{c_i}}{1-p_{v_i}} \tag{9}$$

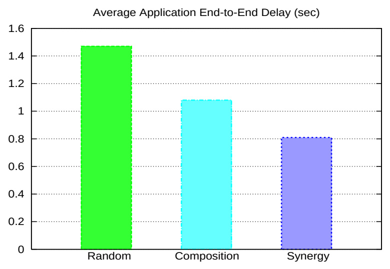
The projected impact δ is acceptable if $\delta + \hat{t} \leq q_t$, in other words if the new projected execution time is acceptable. In the above inequality, q_t is the requested end-to-end execution time QoS metric that was specified by the user in Q_ξ. Similarly to ξ, it is cached for every application 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 [15]. 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 application's probe.

Equation 9 summarizes the QoS Impact Projection algorithm. A Synergy node has locally available all the required information to compute the impact δ for all applications it is currently participating in. This information is available by maintaining local load information, monitoring the local processor utilization, and caching ξ and Q_ξ for all applications it is running, along with their current end-to-end execution times. It uses the projected application execution time to estimate the effect of the component reuse on the existing applications, by considering the effects of increased processor load on the time required to invoke the components.

This projection is performed for all applications currently invoking a component to be reused, for all applications invoking other components located 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. Otherwise, and if there are no other local components that can be reused, the probe is dropped.

4 Experimental Evaluation

We now present the experimental evaluation of Synergy, both through our prototype implementation over the PlanetLab [10] wide-area network testbed, and through simulations. The prototype provided a realistic evaluation. We used simulations in addition to the prototype, to be able to test larger network sizes.



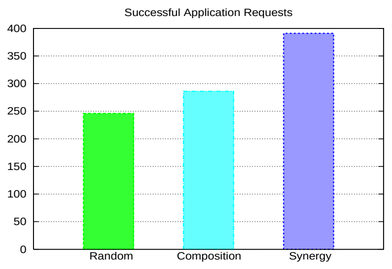


Fig. 6. Average application end-to-end delay. **Fig. 7.** Successful application requests.

4.1 Prototype over PlanetLab

Methodology. Our Synergy prototype was implemented as a multi-threaded system of about 18000 lines of Java code, running on each of 88 physical nodes of PlanetLab. The implementation was based on the SpiderNet service composition framework [13]. Uniformly across the nodes were instantiated 100 components, 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. We generated queries using a Zipf distribution with $\alpha = 1.6$, expecting stream processing applications to follow trends similar to media streaming and web content provision applications [19]. We also experimented with different request distributions in the simulations.

We compared *Synergy* against two different composition algorithms: A *Random* algorithm that blindly selected one of the candidates for each application component. A *Composition* algorithm (such as [13]), that discarded those component candidates whose hosting nodes would not have the required processing power or communication bandwidth to support the request with the specified QoS and among the remaining candidates it chose the ones that resulted in the minimum end-to-end delay.

Results and Analysis. In this set of experiments we investigated Synergy's performance and overhead in a real setting.

Average Application End-to-End Delay. Figure 6 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 streams offers Synergy an advantage, since for some of the requests (fully or partially) only transmission and no processing time is required.

Successful Application Requests. An important metric of the efficiency of a component composition algorithm is the number of requests it manages to accommodate and meet their QoS demands, shown in Figure 7. 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 to applications that fail to meet their QoS demands during their

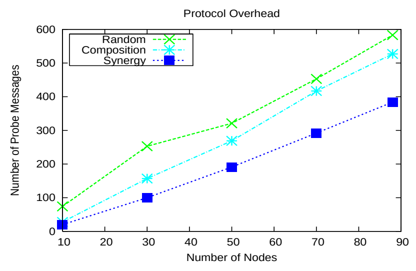


Fig. 8. Protocol overhead.

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

Fig. 9. Breakdown of average setup time.

execution, due to dynamic arrivals of new requests in the system. Synergy's composition algorithm 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 8. 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 to 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 [6]. While Random's overhead could also be tuned to allow less candidates to be visited, its per hop selections would still be QoS-blind.

Average Setup Time. Table 9 shows the breakdown of the average time needed for an application setup, for the three composition algorithms. The setup time is divided in 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. The major part of the time is spent in transmitting probes to candidate components and running the composition algorithm in them. Sharing streams allows Synergy to save time from component probing, which effectively results 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.

4.2 Simulations

Methodology. To further evaluate the performance of Synergy's sharing-aware composition algorithm we implemented a distributed stream processing simulator in about 7500 lines of C++ code. The network topology fed to the simulator was a transit-stub topology of 1500 routers, generated by the GT-ITM internetwork topology generator [20]. We simulated a large overlay network of 500 nodes chosen randomly from

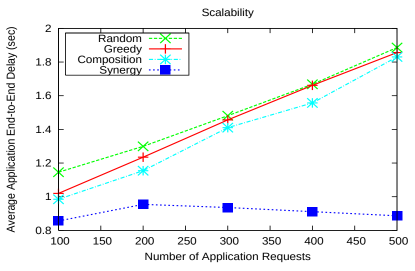


Fig. 10. Scalability.

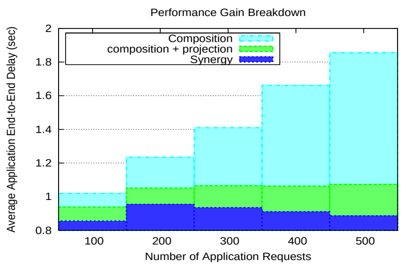


Fig. 11. Performance gain breakdown.

the underlying topology. Nodes and links were assigned processing and communication capacities from discrete classes, to simulate a heterogeneous system.

A total of 1000 components were distributed uniformly across the nodes of the system, with a uniform replication degree of 5. In other words, 200 unique components and 800 component replicas were instantiated at the nodes. Application requests consisted of requests for 2 to 10 components chosen randomly and of streams of random rates transmitted between the components. For each application we set its QoS requirement 30% higher than its projected execution time. We made experiments to investigate both the performance of Synergy's composition algorithm and its sensitivity to the parameters mentioned above.

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 delay between the two components. 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 latency monitoring service such as [17] would be needed. We included it in the simulations though, as a popular centralized approach that provides results with low overhead.

Other than the average application end-to-end delay, which includes processing, transmission, and queueing delays, our main metric for the algorithms' comparison was the success rate, defined as the percentage of application requests that get admitted and complete within their requested QoS limits. This effectively captures the success of a composition algorithm to provide the requested operators, resources, and QoS.

Results and Analysis. In this set of experiments we investigated the performance of Synergy's sharing-aware component composition algorithm for increasing loads.

Scalability. Figure 10 shows the average end-to-end delay of all the applications that are admitted in the system for increasing application load. Synergy consistently achieves the minimum average end-to-end delay. Furthermore, it manages to maintain the average end-to-end delay low, by not admitting more applications than those that can be supported by the system. This is not the case with Random, Greedy, or the Composition algorithm which do not employ QoS impact projection. As the number of deployed and requested applications increases, the probability that existing streams can be shared among applications increases as well. This gives Synergy an additional advantage, which explains the slight decline of the average end-to-end delay for large numbers of application requests.

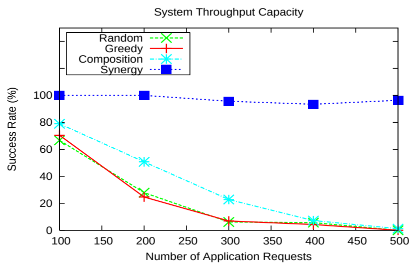


Fig. 12. System throughput capacity.

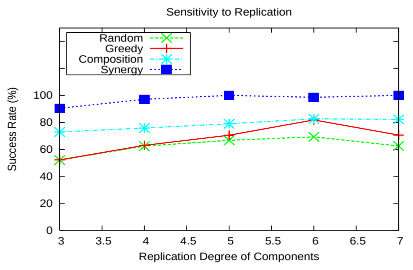


Fig. 13. Sensitivity to replication.

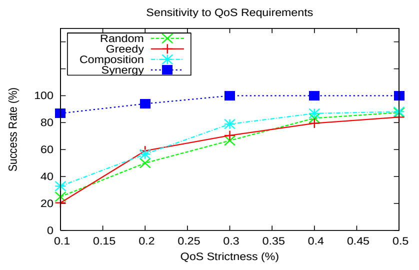
Performance Gain Breakdown. To investigate what part of the performance benefit of Synergy can be attributed to QoS Impact Projection and what part to Maximum Sharing Discovery, we incorporated QoS projection to the Composition algorithm. Figure 11 shows how Composition together with the QoS projection ("composition + projection") compares to Composition and Synergy, in terms of achieved end-to-end delay. QoS projection improves system performance particularly in high loads. While for 100 requests Composition enhanced with projection offers only 8% lower delay than plain Composition, that improvement rises to 42% for 500 requests.

System throughput capacity. Figure 12 shows the success rate for increasing request load. The benefit of sharing-aware component composition is evident, as Synergy is able to scale to much larger workloads, by reusing existing streams. QoS impact projection helps Synergy to achieve very high success rates by avoiding to disrupt currently running applications. Cases of applications that miss their deadlines even with Synergy can be explained by inaccurate estimations because of the current execution time update frequency, or because of inaccuracies in the approximation of the execution time of the admitted applications. As expected, random allocation results in poor QoS. Greedy allocation does not perform well either and the reason is that resources are assigned hop-by-hop ad hoc, blindly to the applications' end-to-end QoS requirements. Another interesting observation is that ensuring that there will be enough resources to run the admitted applications by eliminating resource violations, as the Composition algorithm does, does not suffice for these applications to meet their QoS requirements.

In the following set of experiments we kept the number of application requests at 100, which was a reasonable load for all algorithms as Figure 12 demonstrated. We then investigated the sensitivity of Synergy to various parameters.

Sensitivity to Replication. Figure 13 shows the success rate, as a function of the replication degree of the components in the system. The success of Synergy's composition, as well as its advantage over the other composition algorithms is clear, regardless of the replication degree of the components. Having more candidates to select from in the composition process does not seem to affect the QoS of the composed applications.

Sensitivity to QoS Requirements. Figure 14 shows the success rate as a function of the QoS demands of the applications. Even for very strict requirements, where applications can only tolerate a 10% of extra delay, Synergy's QoS impact projection is able to deliver in-time execution in more than 80% of the cases, whereas the other composition algorithms (Random, Greedy, Composition) fail in as many as 80% of the requests. As QoS requirements become more lax, the performance of those algorithms improves.



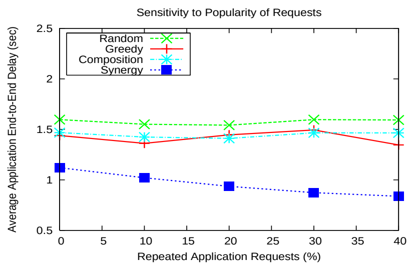


Fig. 14. Sensitivity to QoS requirements. **Fig. 15.** Sensitivity to popularity of requests.

Yet, even in the case of a 50% tolerance in the delay, the best of them, Composition, still delivers 12% less applications within their deadlines than Synergy.

Sensitivity to Popularity of Requests. To investigate how the distribution of user requests affects Synergy's performance in comparison to the rest of the composition algorithms, we assumed a non-Zipfian distribution of application requests with a varying percentage of repetitions. Figure 15 shows the average end-to-end delay of all the applications that are admitted in the system. Synergy utilizes stream sharing and thus can deliver results for the repeated application requests without extra processing. For a request repetition factor of 20% Synergy's Maximum Sharing Discovery algorithm offers 34% lower average end-to-end delay than Composition. For a repetition factor of 40% Synergy achieves an improvement of 25% in comparison to load without any repetitions. Since the rest of the composition algorithms do not offer stream reuse, their performance is not affected by the repetition in application requests. That is as long as the repetition factor is not extremely large, which would result in rejecting application requests due to resource contention.

5 Related Work

Distributed stream processing [4, 9] has been the focus of several recent research efforts from many different perspectives. In [8] and [18] the problem of operator placement 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 operators, rather than deploying new ones. While [8] mentions operator reuse, they do not focus on the impact on already running applications. [7] describes an architecture for distributed stream management that makes use of in-network data aggregation to distribute the processing and reduce the 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 [21], while load balancing based on the correlation of the load distributions across nodes is proposed in [22]. While a balanced load is the final selection criterion among candidate component graphs in Synergy as well, our focus is on QoS provision. The distributed composition probing approach is first presented in [6, 13]. Synergy extends this work by considering stream reuse and evaluating the impact of component sharing. Our techniques for distributed stream processing composition directly apply to multimedia streams [15, 23] as well.

Application task assignment has also been the focus of many grid research efforts. GATES [5] is a grid-based middleware for distributed stream processing. It uses grid resource discovery standards and trades off accuracy with real-time response. While we also address real-time applications, our focus is on the composition of the application component graph. Similarly, work on grid resource management [24] focuses on optimally assigning individual tasks to different hosts, rather than instantiating *composite* network applications. Work on resource discovery such as SWORD [25] can assist in component composition, and is thus complementary to our work.

Component composition has also been studied in the context of web services from many aspects, such as coordinating among different services to develop production workflows [26], or providing reliability through replication [27]. Similar problems are also encountered when providing dynamic web content at large scales [28], or personalized web content [29], the changing and on-demand nature of which render them more challenging than static content delivery [30]. While we focus on component composition for stream processing, our techniques may be applicable to other applications with QoS requirements as well, such as composing QoS-sensitive web services.

6 Conclusion

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 provision for distributed stream processing applications. Our future work includes the integration of iterative execution of Synergy's composition protocol with techniques for application migration. This can enable application adaptation to QoS-affecting changes in the environment, such as a node failure or overload.

References

1. Chandrasekaran, S., et al.: TelegraphCQ: Continuous dataflow processing for an uncertain world. In: Proceedings of CIDR, Asilomar, CA. (2003)
2. Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Datar, M., Ito, K., Motwani, R., Srivastava, U., Widom, J.: STREAM: The Stanford data stream management system. (to appear) (2005)
3. Golab, L., Ozsu, M.: Update-pattern-aware modeling and processing of continuous queries. In: Proceedings of 24th ACM SIGMOD Conference, Baltimore, MD, USA. (2005)
4. Abadi, D., et al.: The design of the borealis stream processing engine. In: Proceedings of CIDR, Asilomar, CA. (2005)
5. Chen, L., Reddy, K., Agrawal, G.: GATES: A grid-based middleware for distributed processing of data streams. In: Proceedings of IEEE HPDC-13, Honolulu, HI. (2004)
6. Gu, X., Yu, P., Nahrstedt, K.: Optimal component composition for scalable stream processing. In: 25th IEEE ICDCS, Columbus, OH. (2005)

7. Kumar, V., Cooper, B., Cai, Z., Eisenhauer, G., Schwan, K.: Resource-aware distributed stream management using dynamic overlays. In: 25th IEEE ICDCS, Columbus, OH. (2005)
8. Pietzuch, P., Ledlie, J., Shneidman, J., Roussopoulos, M., Welsh, M., Seltzer, M.: Network-aware operator placement for stream-processing systems. In: Proc. of 22nd ICDE. (2006)
9. Jain, N., Amini, L., Andrade, H., King, R., Park, Y., Selo, P., Venkatramani, C.: Design, implementation, and evaluation of the linear road benchmark on the stream processing core. In: Proceedings of 25th ACM SIGMOD Conference, Chicago, IL, USA. (2006)
10. PlanetLab Consortium: http://www.planet-lab.org/ (2004)
11. Arabshian, K., Schulzrinne, H.: An ontology-based hierarchical peer-to-peer global service discovery system. Journal of Ubiquitous Computing and Intelligence (JUCI) (2005)
12. Abdelzaher, T.: An automated profiling subsystem for QoS-aware services. In: Proc. 6th IEEE RTAS, Real-Time Technology and Applications Symposium, Washington, DC. (2000)
13. Gu, X., Nahrstedt, K., Yu, B.: SpiderNet: An integrated peer-to-peer service composition framework. In: Proceedings of IEEE HPDC-13, Honolulu, HI. (2004)
14. Rowstron, A., Druschel, P.: Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In: Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms, Heidelberg, Germany. (2001)
15. Chen, F., Repantis, T., Kalogeraki, V.: Coordinated media streaming and transcoding in peer-to-peer systems. In: Proceedings of 19th IPDPS, Denver, CO. (2005)
16. Hu, N., Steenkiste, P.: Exploiting internet route sharing for large scale available bandwidth estimation. In: Proc. of Internet Measurement Conference, IMC, New Orleans, LA. (2005)
17. Tang, C., McKinley, P.: A distributed approach to topology-aware overlay path monitoring. In: Proceedings of 24th IEEE ICDCS, Tokyo, Japan. (2004)
18. Seshadri, S., Kumar, V., Cooper, B.: Optimizing multiple queries in distributed data stream systems. In: 2nd Int. IEEE Workshop on Networking Meets Databases, NetDB. (2006)
19. Cherkasova, L., Gupta, M.: Analysis of enterprise media server workloads: Access patterns, locality, content evolution, and rates of change. IEEE/ACM Transactions on Networking, TON 12(5) (2004) 781–794
20. Zegura, E., Calvert, K., Bhattacharjee, S.: How to model an internetwork. In: Proceedings of IEEE INFOCOM, San Francisco, CA, USA. (1996)
21. Gedik, B., Liu, L.: PeerCQ: A decentralized and self-configuring peer-to-peer information monitoring system. In: Proceedings of 23rd IEEE ICDCS, Providence, RI, USA. (2003)
22. Xing, Y., Zdonik, S., Hwang, J.: Dynamic load distribution in the borealis stream processor. In: Proc. of 21st International Conference on Data Engineering, ICDE, Tokyo, Japan. (2005)
23. Kon, F., Campbell, R., Nahrstedt, K.: Using dynamic configuration to manage a scalable multimedia distributed system. Computer Communications Journal 24 (2001) 105–123
24. Cai, W., Coulson, G., Grace, P., Blair, G., L.Mathy, Yeung, W.: The gridkit distributed resource management framework. In: Proc. of European Grid Conference, EGC. (2005)
25. Oppenheimer, D., Albrecht, J., Patterson, D., Vahdat, A.: Design and implementation tradeoffs for wide-area resource discovery. In: Proceedings of 14th IEEE HPDC-14. (2005)
26. Tai, S., Khalaf, R., Mikalsen, T.: Composition of coordinated web services. In: Proceedings of ACM/IFIP/USENIX 5th International Middleware Conference, Toronto, Canada. (2004)
27. Bartoli, A., Jimenez-Peris, R., Kemme, B., Pautasso, C., Patarin, S., Wheater, S., Woodman, S.: The adapt framework for adaptable and composable web services. IEEE Distributed Systems On Line, Web Systems Section (2005)
28. Amza, C., Cox, A., Zwaenepoel, W.: A comparative evaluation of transparent scaling techniques for dynamic content servers. In: Proceedings of 21st ICDE, Tokyo, Japan. (2005)
29. Colajanni, M., Grieco, R., Malandrino, D., Mazzoni, F., Scarano, V.: A scalable framework for the support of advanced edge services. In: Proc. of HPCC-05, Sorrento, Italy. (2005)
30. Karbhari, P., Rabinovich, M., Xiao, Z., Douglis, F.: ACDN: A content delivery network for applications. In: Proceedings of 21st ACM SIGMOD Conference, Madison, WI. (2002)