UNIVERSITY OF CALIFORNIA
RIVERSIDE

Synergy: Quality of Service Support for Distributed Stream Processing Systems

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Thomas S. Repantis

August 2008

Dissertation Committee:

Professor Vana Kalogeraki, Chairperson
Professor Dimitrios Gunopulos
Professor Michalis Faloutsos

Acknowledgments

I would like to thank my advisor, Prof. Vana Kalogeraki, for the inspiration and guidance, motivation and support she has offered me.

I would also like to thank Prof. Xiaohui Gu for her sharp guidance and inspiring attitude.

I am also grateful to my committee members, Prof. Dimitrios Gunopulos and Prof. Michalis Faloutsos, for their time and their insightful feedback.

I am especially thankful to all my mentors, Dr. Arun Iyengar and Dr. Isabelle Rouvellou from IBM Research, Dr. Michael Kaminsky and Dr. Haifeng Yu from Intel Research, Dr. Debby Levinson and Chris Stroberger from Hewlett-Packard, and Dr. Eric Burger from BEA, for their time and guidance.

I would also like to thank the rest of the members of the Distributed Real-Time Systems lab and the anonymous reviewers of [67, 68, 70–72] for their comments.

Finally, I would like to thank my friends and above all my family for their support throughout these years.

Wenn die Nacht am tiefsten ist, ist der Tag am nächsten.

–Ton Steine Scherben

ABSTRACT OF THE DISSERTATION

Synergy: Quality of Service Support for Distributed Stream Processing Systems

by

Thomas S. Repantis

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, August 2008
Professor Vana Kalogeraki, Chairperson

Many emerging applications in domains such as network traffic management, financial trades surveillance, customized e-commerce applications, and analysis of sensor data, require the real-time processing of large amounts of data that are updated continuously. Distributed stream processing systems offer a scalable and efficient means of in-network processing of such data streams. We propose Synergy, a peer-to-peer middleware that provides Quality of Service support for distributed stream processing applications. Synergy provides sharing-aware component composition, for efficiently reusing data streams and processing components, when composing applications with QoS demands. Utilizing a set of fully distributed algorithms, Synergy discovers and evaluates 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. To proactively identify application hot-spots at run-time, Synergy employs a prediction framework that binds workload forecasting using linear regression with execution time forecasting using correlation. To react to predicted QoS violations and alleviate hot-spots, nodes autonomously migrate the execution of stream processing components using a non-disruptive migration protocol. When deploying new components, Synergy utilizes a decentralized replica placement protocol that aims to maximize availability, while respecting resource constraints, and making performance-aware placement decisions. We have implemented a prototype of the Synergy middleware and evaluated its performance using a real stream processing application operating on real streaming data on PlanetLab, as well as on a simulation testbed. The experimental results demonstrate substantial benefits in performance and QoS provision, while achieving good resource utilization. More information on the Synergy middleware can be found at http://synergy.cs.ucr.edu

Contents

List of Figures

Chapter 1

Introduction

A significant number of emerging on-line data analysis applications require real-time processing of high-volume, high-rate data that are updated continuously, to generate outputs of interest or to identify meaningful events: i) Analysis of clicks or textual input generated by visitors of web sites such as e-commerce stores or search engines, to determine appropriate purchase suggestions or advertisements. ii) Monitoring of network traffic to detect patterns that proclaim attacks or intrusions, to filter out traffic that violates security policies, or to discover trends that can help with network configuration. iii) Customization of multimedia content such as audio or video, provided for entertainment or news coverage, to meet the interests, as well as the hardware and software capabilities of target users. iv) Processing of market data feeds for electronic trading, as well as surveillance of financial trades for fraud detection. v) Monitoring of banking and credit card transactions, or phone records to determine trends or anomalous behavior. vi) Analysis of data collected by sensors monitoring wildlife, plants, traffic, or the environment for fire, earthquakes, or intruders.

We refer to this kind of continuously updated data as streams and the applications that process this kind of data in real-time as stream processing applications [19,54]. Stream processing applications consist of software components, which are processing elements that operate on a set of input streams to produce a set of output streams. A component can offer functionality as simple as filtering, or correlation, or as complex as transcoding, or encryption. Two components are connected if the output of one becomes the input of the other to accomplish the application execution. The combined processing of the data streams by all the different components constitutes the execution of a stream processing application. Thus, stream processing applications are instantiated as directed acyclic graphs connecting components.

Figure 1.1 illustrates a simple stream processing application example from the Stream Query Repository [83]. In this stream processing application example we assume a packet capturing device installed in a network, used by a system administrator that wishes to monitor the source-destination pairs in the top 5 percentile in terms of total traffic in the past 20 minutes over a backbone link. Figure 1.1 shows the components that are involved in processing the packet input over 20-minute windows, to generate the monitoring output.

Because streaming data arrive in high-volumes and high-rates and at the same time the stream processing applications need to produce output within certain time limits to facilitate on-line data analysis, a single machine is often not able to sustain the load of running an entire stream processing application. Therefore, the stream processing components are hosted by different machines, which communicate over a network. We refer to this type of distributed system as a distributed stream processing system (DSPS)

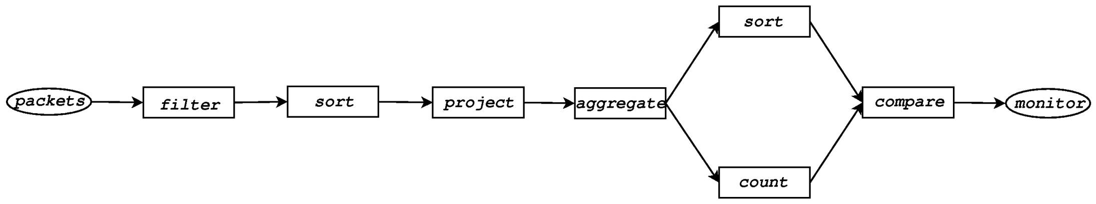


Figure 1.1: Stream processing application example.

(e.g., [1, 6, 21, 35, 46, 61]). The dispersion of the stream processing components of an application among multiple machines is further dictated by the need for reliability. Distributing component replicas on independent machines that are geographically apart increases the application's availability.

Distributed stream processing applications have Quality of Service (QoS) requirements, expressed in terms such as end-to-end delay, throughput, miss rate, or availability. For example, an alert needs to be raised within a certain time frame after an intrusion, or a trading recommendation needs to be made while processing financial data at certain rates. Adhering to such QoS requirements is crucial for the dependable operation of a distributed stream processing system. The first step towards satisfying the QoS requirements of stream processing applications is taking them into account during the application composition. However, as the incoming data rates may increase at run-time, due to external events such as a network attack or a rapid popularity growth of some news event, an application execution may cease to adhere to the requested QoS. Under such unknown and dynamically changing conditions, providing application QoS is a challenging task. The problem is complicated further by the large scale and the distributed nature of a DSPS. Making QoS-affecting decisions at run-time requires detailed monitoring of the system utilization as well as complicated trade-offs that need to be evaluated frequently. Accurate centralized decisions are infeasible, due to the fact that the global state of a large-scale DSPS is changing much faster than it can be communicated to a single host.

In this dissertation we study Quality of Service support for distributed stream processing systems. To achieve Quality of Service we propose techniques for component composition, load balancing, and high availability. We have implemented our techniques in Synergy, a distributed stream processing middleware. Synergy is a software running on every machine of a distributed system to offer distributed stream processing applications, under Quality of Service constraints, and while efficiently managing the system's resources [69, 73]. Unlike previously developed distributed stream processing systems, Synergy offers sharing-aware component composition. Sharing-aware component composition allows stream processing applications that are composed of individual components to utilize previously generated streams and already deployed stream processing components. Synergy's component composition protocol [67, 68] takes the user-requested QoS into account when composing the application component graph. This way, the instantiated application satisfies the QoS requirements defined by the user. Furthermore, Synergy employs hot-spot prediction and alleviation techniques [70, 71], to ensure that the application QoS requirements continue to be met at run-time. Finally, Synergy employs a component placement protocol that aims to maximize application availability [72].

The major research contributions of Synergy can be summarized as follows:

- Synergy employs 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 a new application 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.

- Synergy integrates 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.

- Synergy encompasses a framework built on statistical forecasting methods, to accurately predict QoS violations at run-time and proactively identify application hotspots. In order to achieve this, our prediction framework binds workload forecasting with execution time forecasting. To accomplish workload forecasting we predict rate fluctuations. To achieve this, we exploit auto-correlation in the rate of each component, as well as cross-correlation between the rates of different components of a distributed application. To accomplish execution time forecasting we use the established statistical method of linear regression. This enables us to accurately model the relationship of the application execution time and the entire workload of a node, while dynamically adapting to workload fluctuations.

- Synergy enables nodes to react to predicted QoS violations and alleviate hot-spots by autonomously migrating the execution of stream processing components using a non-disruptive migration protocol. Candidate selection for migration is based on preserving QoS. We employ prediction again to ensure that migration decisions do not result to QoS violations of other executing applications. To drive migration decisions in a decentralized manner we build a load monitoring architecture on top of a Distributed Hash Table (DHT).

- For initial component deployment Synergy employs a decentralized replica placement protocol that aims to maximize availability, while respecting resource constraints, and making performance-aware placement decisions.

We have implemented a prototype of Synergy and evaluated its performance on the PlanetLab [15] wide-area network testbed. We have used a real network monitoring application [83] operating on traces of real TCP traffic [87]. We have also conducted extensive simulations to compare Synergy's composition algorithm to existing alternative schemes. Our experimental results showed that Synergy achieves much better resource utilization and QoS provision than previously proposed schemes, by judiciously sharing streams and processing components during application composition. Our experimental evaluation has also demonstrated high load prediction accuracy, and substantial benefits in application QoS achieved by migration. Finally, our experimental comparison of Synergy's replica placement protocol with the current state of the art has corroborated our claims that Synergy maximizes application availability, while sustaining good performance.

Synergy is implemented as a multi-threaded system of about 35,000 lines of Java code and more information is available at http://synergy.cs.ucr.edu/

We proceed by discussing the contributions of Synergy in more detail, focusing on component composition (Section 1.1), load balancing (Section 1.2), and high availability (Section 1.3). We then present an overview of the rest of the dissertation in Section 1.4.

1.1 QoS-Aware Shared Component Composition

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 it is challenging to select among different component instances to compose stream processing applications on-demand. While previous efforts have investigated several aspects of component composition [35, 46] and placement [61] 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, 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 [6, 35, 46, 61, 94], 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 [33]. 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.

To provide sharing-aware component composition, 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.

Synergy implements 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. Probe messages travel through candidate nodes to determine whether they have enough resources available to accommodate a new application, whether the end-to-end delay achieved is within the required QoS, and whether the impact of the new application would cause QoS violations to existing applications. Synergy utilizes a peer-to-peer overlay for discovering existing streams and components that are also part of a new application request, to avoid redundant computations. Using a maximum sharing discovery algorithm, the graph describing a requested application is backtracked hop-by-hop, to identify up to which point, if any, currently running applications can offer the same results. 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 [36] such as voice-over-IP streams, network traffic and sensor data streams.

Projection is based on queueing theoretical models for both regular and bursty traffic. We approximate bursty traffic with segments of data arrivals of high rate, followed by segments of data arrivals of low rate. To identify the correlation between the segments of different streams, a data arrival time series of each time stream is constructed and maintained, called the signature of the stream, that describes its workload pattern. Stream signatures enable us to combine the processing loads of multiple bursty 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. To reduce network traffic, Synergy collocates new components with their upstream or downstream components based on selectivity.

1.2 Decentralized Hot-Spot Prediction and Alleviation

The first step towards satisfying the QoS requirements of stream processing application is taking them into account during the application composition. However, as the incoming data rates may increase at run-time, due to external events such as a network attack or a rapid popularity growth of some news event, an application execution may cease to adhere to the requested QoS. In fact, the distinct characteristic of stream processing applications is that the data to be processed arrive in high rates, and often in bursts [95, 96]. Under such dynamically changing conditions, providing application QoS is a challenging task. The problem is complicated further by the large scale and the distributed nature of a DSPS.

Synergy addresses the problem of predicting and alleviating application hot-spots in a DSPS. Current approaches for addressing load fluctuations in DSPSs [12, 80, 95, 96], focus on avoiding or resolving hot-spots in the system resources, in other words overloaded nodes. We refer to this kind of hot-spot detection and alleviation as node-oriented. The focus of our work, on the other hand, is on detecting and alleviating hot-spots in the application execution, in other words applications that persistently fail to meet the QoS required by the user. We call this kind of hot-spot detection and alleviation *application-oriented*. We believe that application-oriented hot-spot detection and alleviation are as important as their node-oriented counterparts for the following key reasons: i) Application-oriented hot-spot detection is more sensitive and can be triggered even when a node is underloaded. Even when running on a moderately loaded node, an application may not be meeting its QoS requirements (e.g., if they are stringent), thus experiencing a hot-spot. On the other hand, with node-oriented hot-spot detection, by the time a node is overloaded many of the applications using that node will already have violated their QoS requirements. ii) Application-oriented hot-spot alleviation allows more fine-grained hot-spot alleviation. Depending on the individual applications' QoS demands, only some instead of all the applications that are using a node may be suffering, and thus only these applications may need to be migrated. On the contrary, node-oriented hot-spot alleviation aims at reducing a node's load, irrespective of which of the applications experience overload. iii) Most importantly, application-oriented hot-spot detection enables taking *proactive* measures with regards to application performance, to prevent severe degradation of application QoS.

We propose a framework built on statistical forecasting methods, to accurately predict QoS violations at run-time and proactively identify application hot-spots. In order to achieve this, our prediction framework binds workload forecasting with execution time forecasting. To accomplish workload forecasting we predict rate fluctuations, exploiting auto-correlation in the rate of each component, and cross-correlation between the rates of different components of a distributed application. To accomplish execution time forecasting we use linear regression, an established statistical method, to accurately model the relationship of the application execution time and the entire workload of a node, while dynamically adapting to workload fluctuations.

To react to predicted QoS violations and alleviate hot-spots we enable nodes to autonomously migrate the execution of stream processing components using a non-disruptive migration protocol. Candidate selection for migration is based on preserving QoS. We employ prediction again to ensure that migration decisions do not result to QoS violations of other executing applications. To drive migration decisions in a decentralized manner we build a load monitoring architecture on top of a Distributed Hash Table (DHT) [74].

1.3 Distributed Replica Placement for High Availability

Current DSPSs have been designed to provide low-latency and high-throughput processing of data streams and to adapt to rapid changes in load and resource needs. Another important requirement is the availability of these systems, which is crucial for their correct and continuous operation. To provide high availability, replication of the stream processing components is required. The basic idea behind high availability is that by replicating components and distributing them across different nodes, the failure of a replica will not interrupt the execution of the applications, since other replicas can continue to provide the service. While previous research has shown how component placement affects the performance of distributed stream processing applications [4, 61], in this work we demonstrate how it also affects application availability. We focus on the placement of replicated components to maximize application availability, complementing existing research in the area of high availability for distributed stream processing systems. Existing work in this area has focused on tolerating failures during application execution despite the continuous arrival of data and on fast recovery despite the significant state maintenance overhead. In this context recovery mechanisms [38], failure masking [79], consistency trade-offs [11], and scheduling of checkpoints [18, 39] have been investigated.

Despite its simplicity the example of Figure 1.1 illustrates several key characteristics of distributed stream processing applications, which determine the intricacies in their availability requirements: i) First of all, a distributed stream processing application is composed of several components, as shown in the directed acyclic graph of the example of Figure 1.1. We call such applications *composite*. ii) More importantly, a distributed stream processing application such as the one of Figure 1.1 cannot tolerate availability of a subset of the components. Even if incomplete results can be produced in the absence of certain components, correct execution requires all of them to be available. Therefore, we call such applications *strict*. iii) Moreover, each component may be shared by multiple applications concurrently. For example, the aggregator of Figure 1.1 may be part of more than one monitoring applications. A failure in a shared component has to be masked from all the applications that are currently using it. iv) Furthermore, streaming data typically arrive in large volumes and at high rates, like the traffic in a busy network for the example of Figure 1.1. Failure recovery has to be fast, for the execution to be able to continue with minimal loss. Yet, state maintenance between replicas requires a non-trivial amount of data transfer. v) Finally, even though this is not demonstrated with this simple example, a distributed stream processing application may consist of large numbers of components, distributed over wide-area networks.

Replication for high availability and fault tolerance has been studied from different perspectives in a variety of domains, including distributed databases [28, 60, 63, 91], distributed object systems [26, 27, 31, 45, 57, 64, 93], and web services [14, 51]. While many aspects of replication have been studied extensively, and solutions such as active [76] and passive [17] replication are widely accepted, in this work we focus on replica *placement* for maximizing the availability of distributed stream processing applications. Our placement mechanisms cater to the composite and strict nature of these applications. Ensuring the availability of a composite application differs from guaranteeing the availability of individual objects, such as files in distributed storage systems [3, 5, 42, 75], or databases [28, 60, 63, 91]. Furthermore, the scale of distributed stream processing applications, both in terms of data volume and rate, as well as in terms of numbers of components, affects significantly the placement decisions. For example, not all primary replicas can be hosted by the same server, as might be the case with object-, component-, or service-based architectures, such as CORBA [26, 27, 31, 57], Enterprise JavaBeans [93], or multi-tier architectures [28, 60]. Our placement mechanisms however can be applied to such systems, if their scale requires the primary replicas of a composite application's components to be distributed and significant amounts of data transferred between them make the relative placement of components important.

Synergy addresses the problem of component replica placement to maximize the availability of distributed stream processing applications. We reason and illustrate how the fact that distributed stream processing applications are composite and strict affects the availability of different component replica placements. We then show how the practical constraints in replica placement that arise from the limited processing and network resources available in the system determine the number of nodes to be used for replica placement.

Finally, among placements that are equivalent in terms of availability, we show how to select the one that improves the performance of distributed stream processing applications. While component placement to maximize the performance of distributed stream processing applications has been investigated before [4, 61], to the best of our knowledge this is the first work to discuss component replica placement to maximize the availability of such applications.

We propose a practical and fully distributed component replica placement protocol to implement our design principles. Our protocol collocates components to maximize application availability, respects the processing power and bandwidth availability, and minimizes the communication latency to maximize application performance.

1.4 Dissertation Overview

The rest of the dissertation is organized as follows: Chapter 2 discusses our techniques for taking into account end-to-end delay QoS requirements when composing new applications. Chapter 3 presents Synergy's approach for avoiding end-to-end delay QoS violations during application execution. Chapter 4 discusses Synergy's component replica placement protocol that maximizes availability. Chapter 5 discusses research efforts related to the problems discussed in this dissertation. Finally, chapter 6 presents our conclusions and discusses avenues to future work.

Chapter 2

QoS-Aware Shared Component Composition

2.1 Introduction

In this chapter we discuss how user QoS requirements can be taken into account during the composition of a new application from multiple components. In particular, we present Synergy's sharing-aware component composition protocol [67, 68]. 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.

The rest of the chapter is organized as follows: Section 2.2 introduces the system model and notation used in the rest of the chapter. Section 2.3 discusses Synergy's decentralized sharing-aware component composition approach and its QoS impact projection algorithm. Section 2.4 presents an extensive experimental evaluation of our composition protocol. Finally, brief conclusions are presented in Section 2.5.

2.2 System Model

In this section we present the stream processing application and QoS models that will serve as a basis for our in-depth discussion of Synergy's component composition protocol in the next section (2.3). These models are presented in Section 2.2.1 and Section 2.2.2 respectively. We also provide an overview of the architecture of the prototype implementation of Synergy in Section 2.2.3, as well as an overview of the operation of Synergy's component composition protocol in Section 2.2.4.

2.2.1 Stream Processing Application Model

Table 2.1 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 $\{os_i\}$.

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
$\hat{t}$	Projected End-to-End Execution Time
p_{o_i}	Processing Time Required for o_i
b_{s_j}	Bandwidth Required for s_j

Figure 2.1: Notations.

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). Sources and destinations are components that are typically pinned on specific nodes and act as initial producers and end-consumers of streams respectively. 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 2.6, the name of s_2 is $o_1(s_1)$, to reflect 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 *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[1]. 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 2.3.1.

2.2.2 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

[1]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 work. Thus, we assume the query plan is given.

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 [2]. 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.

The monitoring module of each Synergy node v_i is responsible for maintaining resource utilization information for v_i and the virtual links connected to v_i. In particular, 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 [56] or [37]). We also use b_{l_j} to denote the amount of current bandwidth consumed on l_j.

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., $q_t^\lambda \leq q_t^\xi$.

2.2.3 Synergy Architecture

We now present a brief overview of Synergy's architecture. The middleware's goal is to support the execution of distributed stream processing applications with QoS constraints, while efficiently managing the system's resources. Synergy adopts a fully distributed architecture, where any node of the middleware can request or participate in a distributed stream processing application. In Synergy, data streams, consisting of independent data tuples, arrive continuously from external sources (such as web users, monitoring devices, or a sensor network) and need to be processed by stream processing components in real-time. Each component is a self-contained software module, that offers a predefined operator. The operators can be as simple as a filter or a join, or as complex as transcoding or encryption. Components are deployed in the distributed nodes of the Synergy middleware according to their individual software capabilities or following criteria for the optimization of the performance of the whole system [4, 59, 61, 78].

Each Synergy node is identified by its IP and port. The nodes of our distributed stream processing middleware are connected via overlay links on top of the existing IP network as shown in Figure 2.2. To facilitate decentralized component discovery, we organize the peers in a Distributed Hash Table (DHT), currently FreePastry [74]. Utilizing a DHT structure enables us to efficiently locate components and load information about the nodes. The application component graph is built on top of the middleware, as shown in Figure 2.2.

As illustrated in Figure 2.2, each node of the middleware consists of the following main modules: i) A *discovery module* that is responsible for locating existing data streams and components. Synergy leverages the structure of the underlying overlay network for reg-

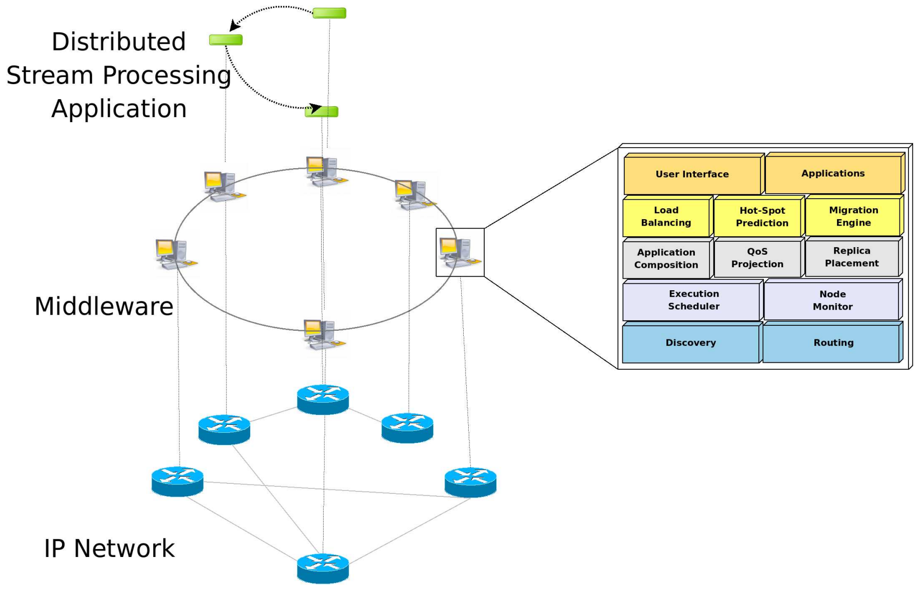
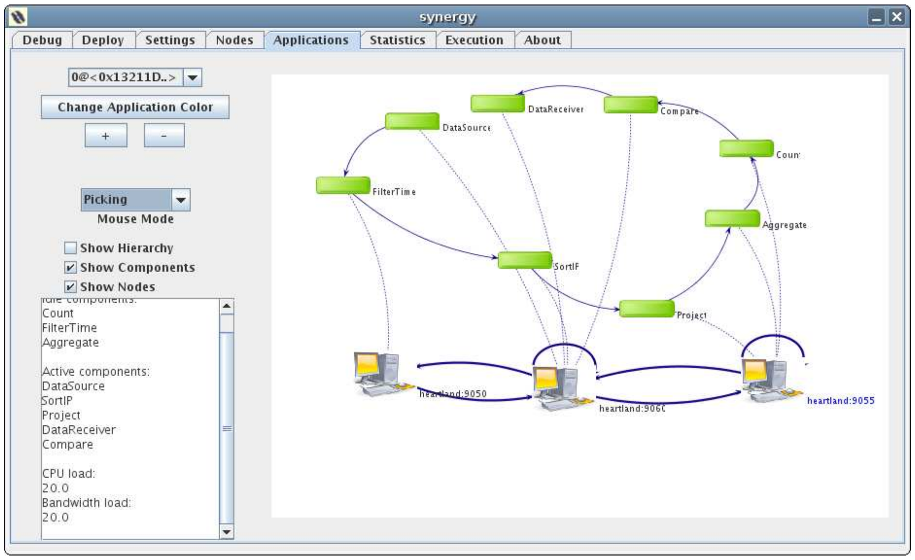


Figure 2.2: Synergy system architecture.

istering and discovering available components and streams in a decentralized manner. In our current prototype we implement a keyword-based discovery service, on top of the Pastry distributed hash table (DHT) [74]. This allows us to register and discover components by hashing keywords instead of the component IDs themselves, and thus decouple component placement from their discovery. ii) A *routing module* that routes protocol messages between Synergy nodes through the overlay, and data streams either through the overlay, or by opening direct TCP connections. iii) A *monitoring module* that is responsible for maintaining resource utilization information for the node and the virtual links connected to it. In the current implementation, the monitoring module keeps track of the CPU load, the network bandwidth, and latencies to other nodes. iv) A *scheduling module* that implements various algorithms to schedule the execution of the stream processing applications running on a node. v) An *application composition module* that composes stream processing applications



Idle components:
Count
FilterTime
Aggregate

Active components:
DataSource
SortIP
Project
DataReceiver
Compare

CPU load:
20.0
Bandwidth load:
20.0

Figure 2.3: Application instantiation on Synergy.

from already deployed components. The middleware adopts a fully decentralized architecture, where any node can compose a distributed stream processing application [67, 68]. vi) A *QoS projection module*, which is used during application composition to determine the impact of admitting new applications to the QoS of currently running ones [67, 68]. vii) A *replica placement module*, that determines on which nodes to place the component replicas of a composite application to maximize application availability [72]. viii) A *load balancing module* that proactively alleviates hot-spots relying on decentralized migration decisions [70, 71]. ix) A *hot-spot prediction module* that drives load management decisions by predicting QoS violations at run-time [71]. x) A *migration engine* that enables Synergy nodes to autonomously migrate the execution of stream processing components using a non-disruptive migration protocol [70, 71]. xi) An *application module* that implements the logic of the various stream processing applications the middleware can offer. We have currently implemented network traffic monitoring and stream encryption. xii) A *user interface module* to control and monitor the execution of the middleware and of the applications. The user interface includes a command line interface and the graphical interface shown in Figure 2.3, demonstrating our implementation of the network traffic monitoring application illustrated in Figure 1.1.

2.2.4 Approach Overview

We now give a brief overview of Synergy's component composition. 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, the system first tries to discover whether any of the requested streams have been generated by previously instantiated query plans. To

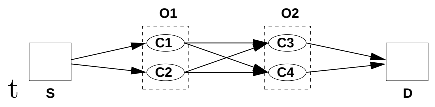


Figure 2.4: Probing example.

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 operators in the query plan 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 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 2.4 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.

2.3 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 composition. 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 component deployment, stream reuse and component sharing. Synergy's fully distributed and light-weight composition protocol is executed when instantiating a new application.

2.3.1 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

```
**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
        sends acknowledgement message to upstream node
        performs transient resource reservation at $v_i$
        discovers next-hop candidate components from $\xi$
        deploys next-hop candidate components if needed
        spawns probes for selected components
    **else**
        drops received probe
$v_s$ selects most load-balanced component composition $\lambda$
$v_s$ establishes stream processing session
```

Figure 2.5: Synergy composition algorithm.

generated query plan ξ and then instantiates the application component graph based on the user's QoS requirements Q_ξ. Figure 2.6 shows an example of a query plan, while Figure 2.7 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., [32, 35]) 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 Figure 2.5, 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 2.6, 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 2.3.3. Next, the Synergy node v_s initiates a distributed probing process to collect resource and QoS states from those candidate components that

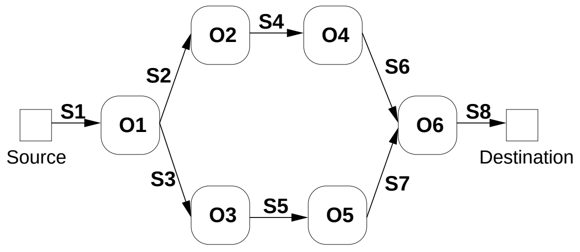


Figure 2.6: Query plan example.

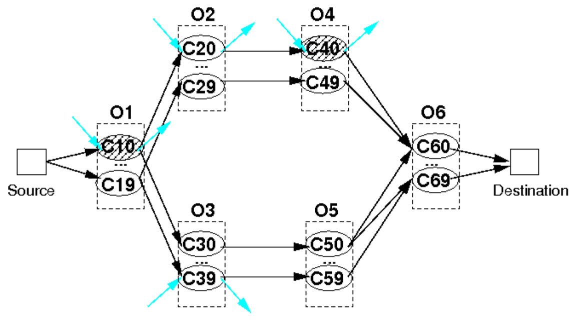


Figure 2.7: Synergy composition example.

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 [35]. 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.

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[2]. 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 [2]. 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 2.3.4. 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.

[2]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.

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 [47] 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 2.3.2. 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 2.7, 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:

$$operator\ \ constraints : c_i.o = o_i,\ \forall o_i \in \xi, \exists c_i \in \lambda \tag{2.1}$$

$$QoS\ \ constraints : q_r^\lambda \leq q_r^\xi, 1 \leq r \leq m \tag{2.2}$$

$$processing\ \ capacity\ \ constraints : rp_{v_i} \geq 0, \forall v_i \in V_\lambda \tag{2.3}$$

$$bandwidth\ \ constraints : rb_{l_j} \geq 0,\ \ \forall l_j \in \lambda \tag{2.4}$$

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:

$$\phi(\lambda) = \sum_{v_i \in V_\lambda, o_i \in \xi} \frac{p_{o_i}}{rp_{v_i} + p_{o_i}} + \sum_{l_j \in \lambda, s_j \in \xi} \frac{b_{s_j}}{rb_{l_j} + b_{s_j}} \tag{2.5}$$

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 [4, 59, 61, 78]. 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 work. 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 [32] could be adopted to integrate our solution with utility-based approaches [20], 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.

2.3.2 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 2.8: i) If the missing

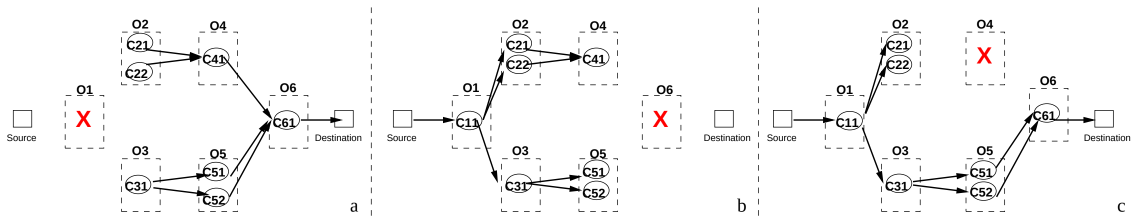


Figure 2.8: The three cases of new component deployment.

component is at the beginning of the graph, we can collocate it with any of its downstream candidates. Thus, in Figure 2.8.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 2.8.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 2.8.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 [90], 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 2.8.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 2.8.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 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 [47]. For example, in Figure 2.8.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.

2.3.3 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 **??**, the current implementation is built over Pastry [74]. 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 node hosting the component or stream. As was described in Section 2.2.1, 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 2.6 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 2.6, 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.

2.3.4 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 2.7, 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 2.3.4) and then for bursty traffic (Section 2.3.4).

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 queue as it waits for other components to complete[3]. 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.

[3]Although different scheduling algorithms will determine different orders for the execution of the components at each node, we assume here an FCFS scheduling order.

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 [43]. 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 [8]. Our experimental results show that this simplified model can provide good projection performance for both synthetic and real datasets. tIf 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:

$$x_{c_i,v_i} = \frac{\tau_{c_i}}{1 - p_{v_i}} \tag{2.6}$$

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:

$$y_{s_j,l_j} = \frac{\sigma_{s_j}}{1 - b_{l_j}} \tag{2.7}$$

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:

$$\hat{t} = max_{path} \sum_{v_i \in V_\lambda, l_j \in \lambda} \left(\frac{\tau_{c_i}}{1 - p_{v_i}} + \frac{\sigma_{s_j}}{1 - b_{l_j}} \right) \tag{2.8}$$

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 [2]. 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[4]:

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

[4]We currently calculate the impact projection based on the projected execution time, as the projected communication time may not be accurate in a shared network such as the Internet, where virtual links correspond to multiple physical links shared with extraneous traffic.

time, as the difference between the projected end-to-end execution time after the reuse, $\hat{t}'$, and the execution time before the reuse, $\hat{t}$:

$$\delta = \hat{t}' - \hat{t} = \frac{\tau_{c_i}}{1 - (p_{v_i} + p_{c_i})} - \frac{\tau_{c_i}}{1 - p_{v_i}} \tag{2.9}$$

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. 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 [20]. 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.

QoS Impact Projection for Bursty Traffic

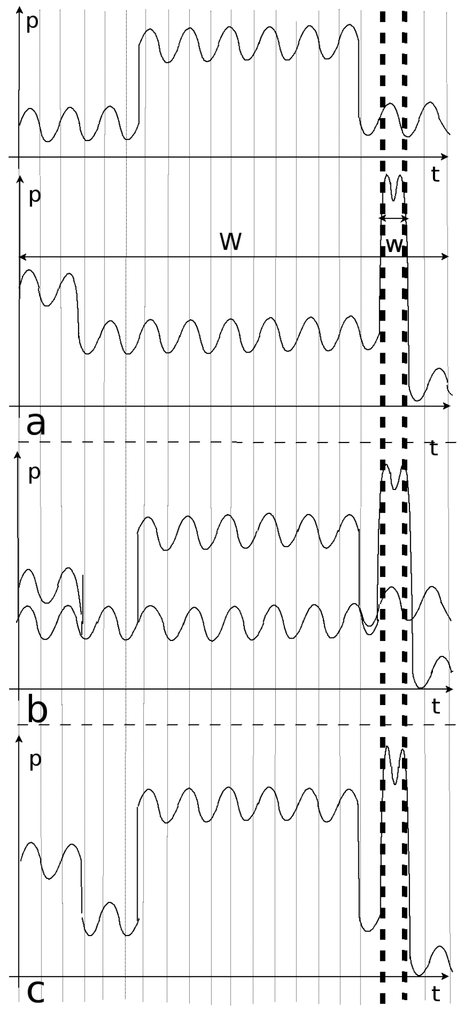
Oftentimes streaming data, such as voice-over-IP data, network traffic, or sensor measurements generated in an emergency application, can be bursty, and therefore well approximated by an ON/OFF model [9,36]. In an ON/OFF model, segments of data arrivals with high rate are followed by segments of data arrivals with low rate. Similar to [9, 36], 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 2.3.4. 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 [34]. 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. Sig-



!t

Figure 2.9: The two stream signatures shown in a) are aggregated as in b) to get the combined signature of c).

natures 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.

As Figure 2.9 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 2.9. 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 2.9, 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 [43]. 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:

$$x_{c_i,v_i,w} = \frac{\tau_{c_i}}{1 - \sum\limits_{\Omega_{j,w} \forall s_j \in S_{v_i}} p_{\Omega_{j,w}}} \tag{2.10}$$

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: $\frac{\tau_{c_i}}{1-(\sum\limits_{\Omega_{j,w} \forall s_j \in S_{v_i}} p_{\Omega_{j,w}} + p_{c_i})}$. 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 also 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 the reuse, $\hat{t}$:

$$\delta_w = \hat{t}' - \hat{t} = \frac{\tau_{c_i}}{1 - (\sum\limits_{\Omega_{j,w} \forall s_j \in S_{v_i}} p_{\Omega_{j,w}} + p_{c_i})} - \frac{\tau_{c_i}}{1 - \sum\limits_{\Omega_{j,w} \forall s_j \in S_{v_i}} p_{\Omega_{j,w}}} \tag{2.11}$$

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 2.9 and 2.11 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 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.

2.4 Experimental Evaluation

We now present the experimental evaluation of Synergy, both through our prototype implementation over the PlanetLab [15] wide-area network testbed, and through simulations. The prototype showed the feasibility and practicality of our approach. Additionally, we used simulations to perform more extensive experiments.

2.4.1 Prototype over PlanetLab

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 [15]. The implementation was based on the SpiderNet service composition framework [32]. 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 [22]. 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 [32]), that performs QoS-aware component composition but does not consider result stream reuse or the effects of component reuse on the applications' QoS.

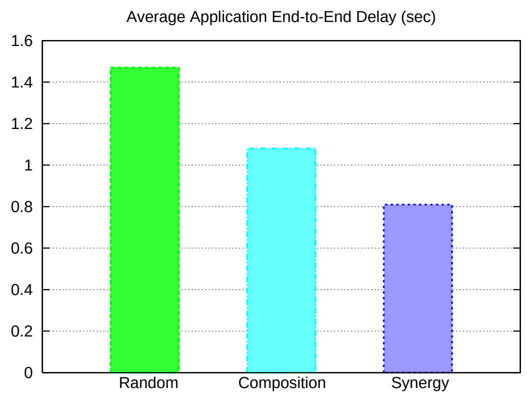


Figure 2.10: Average application end-to-end delay.

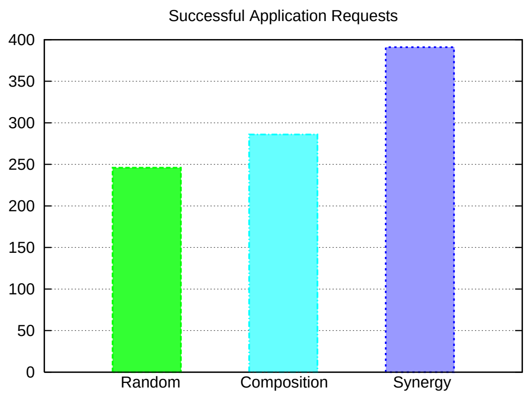


Figure 2.11: Successful application requests.

Results and Analysis

Average Application End-to-End Delay. Figure 2.10 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 2.11. 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 2.12. 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 [35]. 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 2.13 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 the composition algorithm. Sharing

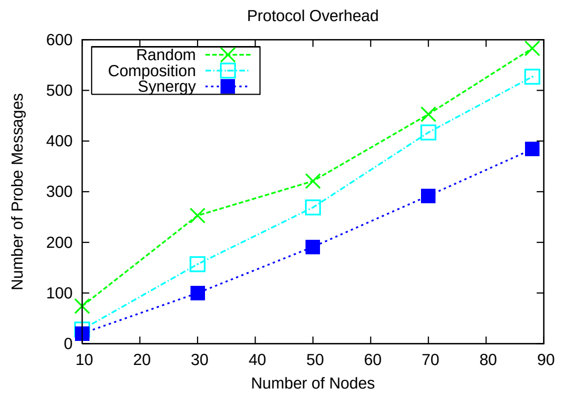


Figure 2.12: Protocol overhead.

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

Figure 2.13: Breakdown of average setup time.

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.

2.4.2 Simulations

Methodology

To further evaluate the performance of Synergy's sharing-aware composition algorithm, we implemented a distributed stream processing simulator in about 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 [100]. 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.

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 delay monitoring service such as the ones discussed in [47] would be needed. 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

Scalability. Figure 2.14 shows the average end-to-end delay of all the applications that

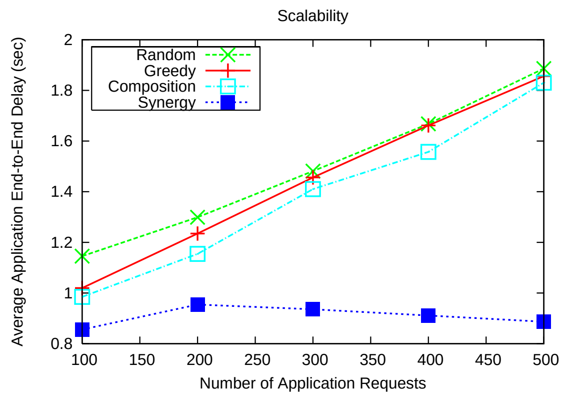


Figure 2.14: Scalability.

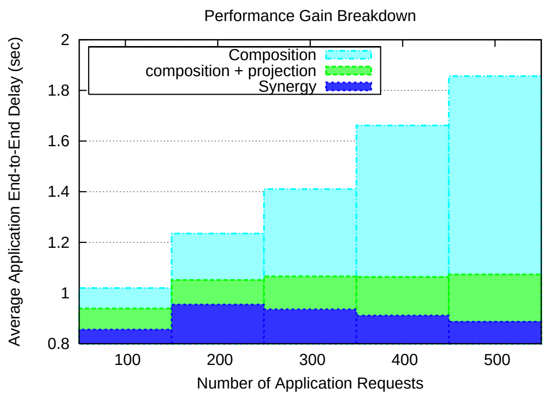


Figure 2.15: Performance gain breakdown.

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 requests.

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 2.15 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 2.16 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 disrupting 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. The Greedy algorithm does not perform well either, because the per hop greedy component selection does not necessarily lead to the best end-to-end composition. Another interesting observation is that there is a significant number of QoS violations due to component reuse, which are not handled by the Composition algorithm.

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 2.16 demonstrated. We then investigated the sensitivity of Synergy to various parameters.

Sensitivity to Replication. Figure 2.17 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.

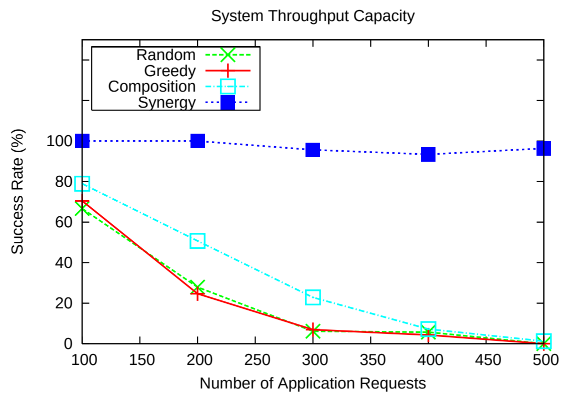


Figure 2.16: System throughput capacity.

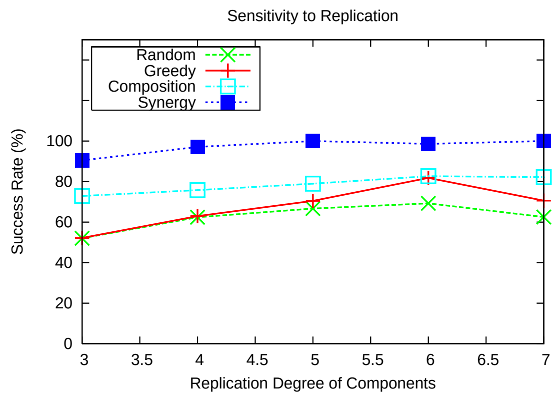


Figure 2.17: Sensitivity to replication.

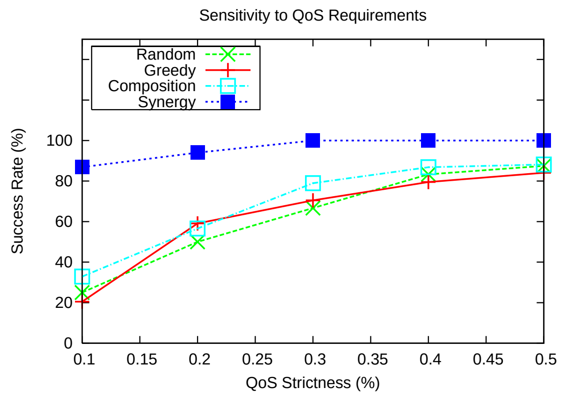


Figure 2.18: Sensitivity to QoS requirements.

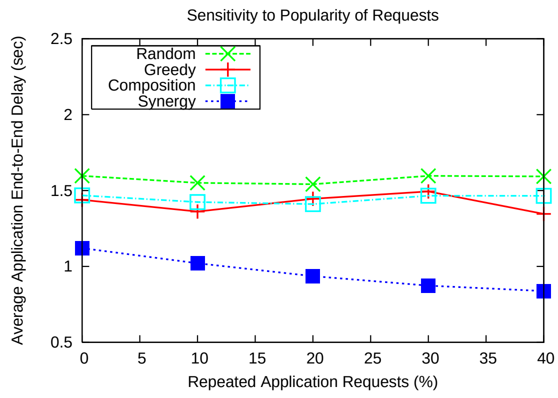


Figure 2.19: Sensitivity to popularity of requests.

Sensitivity to QoS Requirements. Figure 2.18 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. 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 2.19 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.

In the next set of experiments, we investigated the performance benefit of Synergy's collocation-based component deployment strategy, described in Section 2.3.2.

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 2.4.2. Synergy collocated the new component with another component of the application component graph, based on the heuristics described in Section 2.3.2, also performing the required resource and QoS checks. Composition and

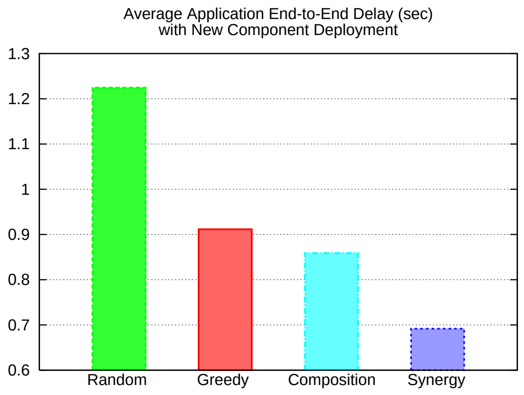


Figure 2.20: Average end-to-end delay with deployment.

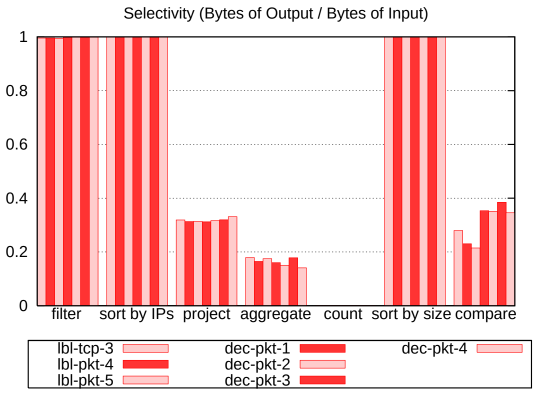


Figure 2.21: Selectivity of different operators.

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 2.20 shows the average application end-to-end delay achieved by the different algorithms. 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 [4,59,61,78]. Figure 2.20 shows average end-to-end execution delay after the application has been instantiated, for 100 application requests. The comparison of this end-to-end delay with the one achieved when all components are being reused and no new component is deployed, which was shown in Figure 2.14, leads to the following observation: While delay-based deployment decreases delay by 12% due to the increase in available choices for component hosts, collocation-based deployment still provides a more substantial decrease of 21%.

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[5] from the stream query repository [83] and recorded the output of the operators for streams produced by seven different traces of network traffic from the Internet traffic archive [87]. Figure 2.21 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 final set of experiments we examined the accuracy of Synergy's QoS impact projection algorithm described in Section 2.3.4. 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.

[5]Application screenshots available at http://synergy.cs.ucr.edu/screenshots.html

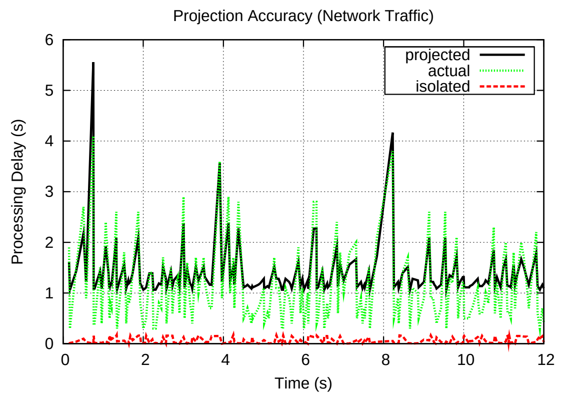


Figure 2.22: Projection accuracy for network traffic.

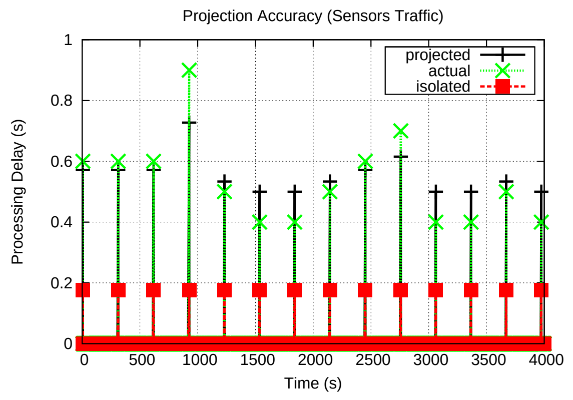


Figure 2.23: Projection accuracy for sensor traffic.

Projection accuracy for real network traffic. We investigated 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 [87]. 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 2.22, 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 [89]. 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

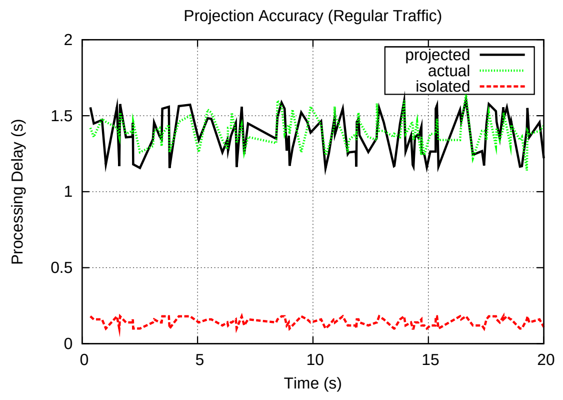


Figure 2.24: Projection accuracy for regular traffic.

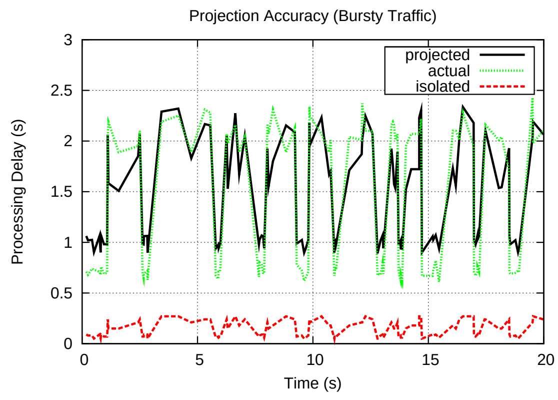


Figure 2.25: Projection accuracy for bursty traffic.

humidity. A burst of measurements lasting approximately one second was generated every five minutes. Figure 2.23 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 2.24 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 2.25 shows that Synergy's projection algorithm accurately identifies the high and low rate segments. Similar to the projection for network traffic of Figure 2.22, the projections for the low rate segments are mostly conservative, i.e., above the actual delays. However,

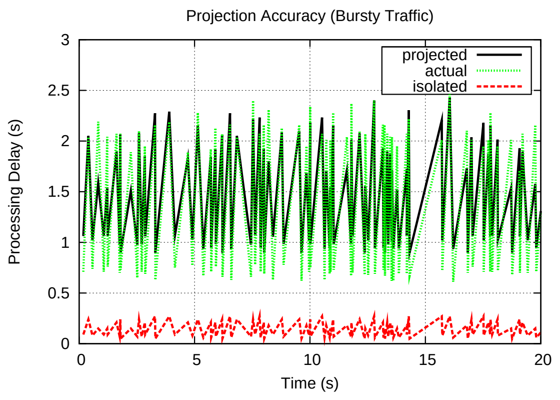


Figure 2.26: Projection accuracy for bursts with period 0.5 second.

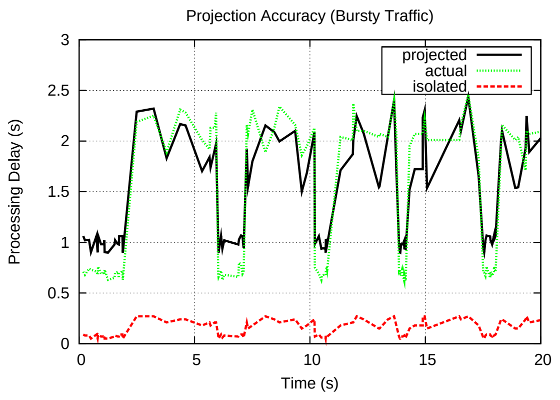


Figure 2.27: Projection accuracy for bursts with period 5 seconds.

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 2.25.

Sensitivity to burst period. Figures 2.26 and 2.27 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 2.28 and 2.29 show the projection accuracy when varying the burst rate. Figure 2.28 shows traffic with high rate of 3 tuples/second and low rate of 1 tuple/second, while Figure 2.29 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 2.25, which

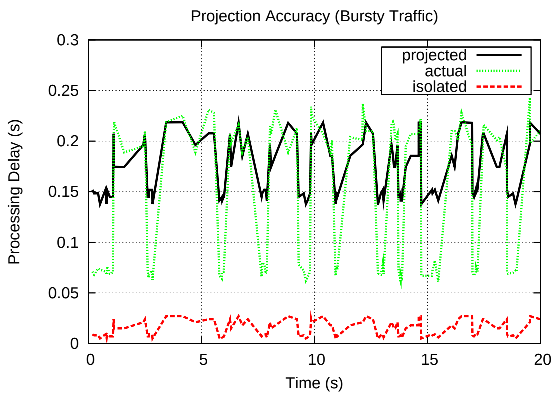


Figure 2.28: Projection accuracy for bursts with high rate 3 tuples/s and low rate 1 tuple/s.

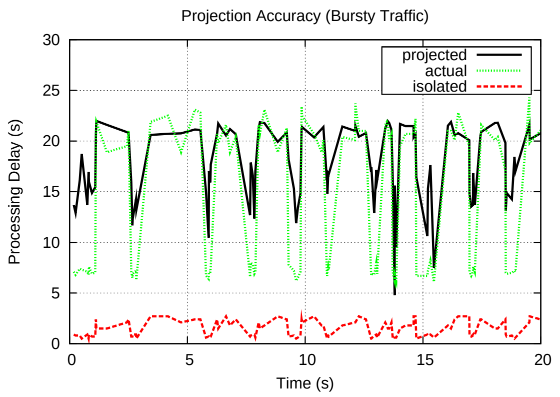


Figure 2.29: Projection accuracy for bursts with high rate 300 tuples/s and low rate 100 tuples/s.

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 2.30 and 2.31 show the projection accuracy when varying the ratio of the rates of the high- and low-rate segments. Figure 2.30 shows traffic with high rate of 20 tuples/second and low rate of 10 tuples/second, while Figure 2.31 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.

2.5 Conclusions

In this chapter we have presented Synergy's sharing-aware component composition algorithms. Following a totally decentralized architecture, Synergy utilizes a Maximum Sharing Discovery algorithm to reuse existing streams, and a QoS Impact Projection

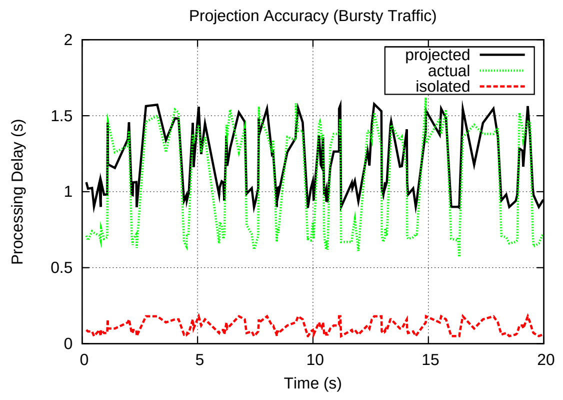


Figure 2.30: Projection accuracy for bursts with high rate 20 tuples/s and low rate 10 tuples/s.

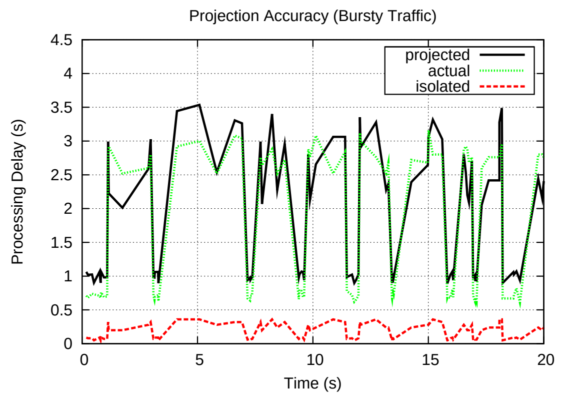


Figure 2.31: Projection accuracy for bursts with high rate 40 tuples/s and low rate 10 tuples/s.

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.

Chapter 3

Decentralized Hot-Spot Prediction and Alleviation

3.1 Introduction

In the previous chapter we discussed how user QoS requirements can be taken into account when composing a new application. We now discuss the problem of adhering to QoS requirements while an application is executing. The large scale and the distributed nature of stream processing systems, as well as the fluctuation of their load, render it difficult to ensure that distributed stream processing applications meet their QoS demands. We describe a decentralized framework for proactively predicting and alleviating hot-spots in distributed stream processing applications in real-time [70,71], as well as its implementation and evaluation in Synergy, our distributed stream processing middleware. Resource monitoring, and hot-spot prediction and alleviation are carried out by all nodes independently, building upon a completely decentralized architecture. We base our hot-spot prediction techniques on the statistical forecasting methods of linear regression and correlation. To alleviate hot-spots, Synergy empowers nodes to autonomously migrate the execution of stream processing components using a non-disruptive migration protocol.

The rest of this chapter progresses as follows: Section 3.2 introduces the system model used in the rest of the chapter. Building upon this model, we describe Synergy's autonomic load management techniques. We treat application hot-spots in the Synergy distributed stream processing middleware in two steps: First, we proactively predict their occurrence by using statistical prediction methods (Section 3.3). Second, we alleviate them by taking local migration decisions (Section 3.4). The experimental evaluation of our approach is presented in Section 3.5, while Section 3.6 presents brief conclusions.

3.2 System Model

In this section we present the stream processing application model used in Synergy. This serves as a background for our discussion of Synergy's load management techniques presented in the next sections.

The user executes a distributed stream processing application by submitting a request at one of the peers of the middleware, specifying the required operators and their dependencies. Then, the system runs the composition algorithm we described in the previous chapter to select the components on the peers to accomplish the application execution. The composition algorithm takes into account the application QoS requirements and resource availability and specifies the components that will constitute the application component graph. Once the components have been selected, each peer hosting one of these components is notified, so that the execution can begin. Once a peer receives such a notification, it invokes the particular component. Each component participating in an application execution is aware of its upstream components that supply it with data, and its downstream components, to which it sends processed data.

We have extended Synergy's architecture to enable decentralized load monitoring, built on top of the DHT we use for component discovery [74]. We describe the details of Synergy's load monitoring architecture in Section 3.4.3. We use our decentralized load monitoring architecture to cope with application hot-spots. We define an application hot-spot as a node in the application component graph in which the application execution persistently fails to meet the QoS required by the user. The end-to-end QoS requirements, which are specified when requesting an application, may among others include end-to-end execution time, throughput, or miss rate. Although our schemes are generic to additive QoS metrics linearly related to rate, we focus on the end-to-end execution time metric denoted by q_t.

3.3 Application Hot-Spot Prediction

The goal of proactive application hot-spot detection is to predict end-to-end execution time QoS violations. In order to achieve this goal we employ: i) Computation of the application "slack time" t_s (Section 3.3.1), to determine the maximum local execution time allowed by the application QoS, before missing its end-to-end execution time requirement. ii) Local execution time prediction based on an application's incoming rate and using linear regression, to determine whether the maximum local execution time will be reached or exceeded (Section 3.3.2). iii) Rate prediction based on auto- and cross-correlation between stream processing components, to determine the future workload that defines the future execution time (Section 3.3.3).

3.3.1 End-to-End to Local Execution Time Translation

We predict an application hot-spot by examining the "slack time" of the application on every component of the application component graph. The slack time represents how close we are to violating the end-to-end execution time requirement of the application. Let q_t represent the end-to-end execution time requirement of the application. q_t includes the execution and communication times spent for a tuple to traverse the entire application component graph. Thus, we define the slack time t_s of an application as the difference between the required end-to-end execution time q_t and the predicted end-to-end execution time. As the application executes, its slack time is computed for every tuple, on every component of the application component graph, based on the local prediction of the end-to-end execution time. The predicted end-to-end execution time includes the execution and communication times spent for a tuple to reach the current component, t_e and t_c respectively, the predicted execution times $\hat{t_e}$ needed for the current and its downstream components to process the data tuple, as well as estimated average communication times $\bar{t_c}$ needed for the data tuple to traverse the rest of the application component graph. For example, in Figure 3.2 the predicted end-to-end execution time as it is calculated in component B is the sum of $t_{e(A)}$, $t_{c(A \to B)}$, $t_{e(\hat{B})}$, $t_{c(\bar{B} \to D)}$, and $t_{e(\hat{D})}$. In order to avoid a QoS violation, the predicted end-to-end execution time needs to be less than the required end-to-end execution time q_t, in other words, the slack time t_s needs to be positive, for every component i of the v components of the application component graph:

$$\begin{aligned} t_{s(i)} = q_t - (\sum_{j \in 1 \ldots i-1} t_{c(j \to j+1)} + \sum_{j \in 1 \ldots i-1} t_{e(j)} + \\ \sum_{j \in i \ldots v-1} \bar{t}_{c(j \to j+1)} + \sum_{j \in i \ldots v} \hat{t}_{e(j)}) > 0 \end{aligned} \tag{3.1}$$

The above single-path computation will identify a hot-spot in the path where it exists. For example, if in Figure 3.2 component C is overloaded, the path $A \to B \to D$ will not detect a hot-spot, while path $A \to C \to D$ will. In order for the above hot-spot prediction to take place, the estimated average communication times, and the predicted execution times must be computed. The estimates for the communication times are available from the application composition phase [68] and can be updated periodically. The predicted execution times are derived locally on every node hosting a component of the application component graph, as explained in the following Section 3.3.2. They are then propagated to all nodes participating in the application execution using a feedback loop passing through the source. The feedback loop allows us to piggyback the predicted execution times on the data tuples, to minimize the communication overhead. For example, in the application component graph shown in Figure 3.2 when the node hosting component D calculates the component's next predicted execution time for this application, it propagates it to the node hosting component A, which forwards it to the nodes hosting components B and C. Similarly, the rest of the nodes propagate their predicted execution times. Using the predicted execution times to compute the slack time on every component enables us to predict locally whether the end-to-end execution time requirement of the application will be violated.

3.3.2 Local Execution Time Prediction

In this section we explain how we predict the local execution time $\hat{t_e}$ needed to process a data tuple of an application. The prediction takes place at each node hosting a component of the application. $\hat{t_e}$ is used to compute the next slack time t_s of the application using Equation 3.1. The local execution time for a data tuple (the time elapsed between the arrival and the departure of the tuple) is the sum of the processing time to process the tuple, and the queueing time the tuple has to wait in the scheduler's queue while other tuples are being processed. While the processing time is constant for a given tuple size, the queueing time depends on the load of the processing node, in other words on the rates (incoming tuples to be processed per time unit) and processing times of the applications currently being executed on the node. Using queueing theory, one can derive average values for the queueing time, assuming an M/M/1 queueing model [68], or a more general M/G/1 model that makes no assumptions regarding the service rate, in which case the queueing time is given by the Pollaczek-Khinchin mean value formula [43]. However, we chose not to predict the execution time using queueing theory for the following reason: The arrivals of data tuples may not always be accurately approximated with a Poisson distribution if rate fluctuations or bursts occur. Such rate variations are quite common in distributed stream processing applications [95]. Accurate prediction during such fluctuations is however crucial. We use linear regression to predict the execution time of an application [53]. Since data tuples arrive in high rates, prediction is more fine-grained than node load changes.

To predict the local execution time $\hat{t_e}$ of an application using a component on a node, we need to derive the relationship between $\hat{t_e}$ and the total rates $r_t = \sum_{l \in 1 \ldots a} r_l$ of all a applications currently using components on that node. While for increasing r_t one expects $\hat{t_e}$ to increase, the trend of the increase is not clear without making any assumptions regarding the arrival pattern of the data tuples. We approximate the relationship using linear regression and our experimental results show good fitting for increasing rates. Figures 3.16, 3.17 show the relationships between the execution times of different components of a stream processing application and the rates of the applications currently running on the nodes hosting them, obtained from our implementation over Planetlab. Linear relationship of execution time and rate is also consistent with earlier works [90, 95].

Each node maintains a series of (t_e, r_t) pairs, for each application a component of which the node is hosting. The series is maintained as a sliding window of the k most recent values. The execution time is measured every time a data tuple for an application is processed, while the total rate is measured as the sum of rates of all applications, data tuples of which were processed since the last time a data tuple of that application was processed. If the rate of any application increases, it affects the execution time of other applications on the same node due to queueing delays. We estimate the conditional expected value of t_e, given a predicted value for r_t. We use linear regression, and assuming we have k pairs so far, the linear function is $t_e = a + b \cdot r_t$ and the least square estimators a and b are:

$$a = \bar{t_e} - b \cdot \bar{r_t} \qquad b = \frac{\sum_{j \in 1 \ldots k} (r_{t(j)} - \bar{r_t}) \cdot (t_{e(j)} - \bar{t_e})}{\sum_{j \in 1 \ldots k} (r_{t(j)} - \bar{r_t})^2} \tag{3.2}$$

where the average values $\bar{t_e}$ and $\bar{r_t}$ are:

$$\bar{t_e} = \frac{\sum\limits_{j \in 1 \ldots k} t_{e(j)}}{k} \qquad \bar{r_t} = \frac{\sum\limits_{j \in 1 \ldots k} r_{t(j)}}{k} \tag{3.3}$$

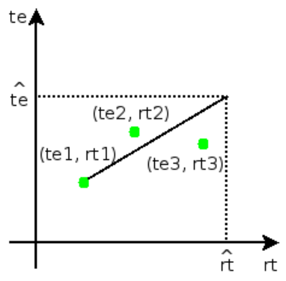


Figure 3.1: Linear regression for local execution time prediction.

In order to enable proactive hot-spot detection, we base the prediction of the execution time $\hat{t_e}$ of an application on the predicted rates of the applications running on components of the node, $\hat{r_t} = \sum\limits_{l \in 1 \ldots a} \hat{r_l}$. (We explain how $\hat{r_l}$ for an application l is derived in the following Section 3.3.3.) Assuming an estimated value for the next $\hat{r_t}$, we predict $\hat{t_e}$ using the above equations. Specifically, as shown in Figure 3.1, we use the k pairs of (t_e, r_t) values to calculate a and b and then given an estimated $\hat{r_t}$ we predict $\hat{t_e}$ using the following formula:

$$\hat{t_e} = a + b \cdot \hat{r_t} \tag{3.4}$$

To evaluate the accuracy of our execution time prediction we calculate the estimated standard error of the slope b:

$$se(b) = \sqrt{\frac{\sum\limits_{j \in 1 \ldots k} (t_{e(j)} - \bar{t_e})^2 - b \sum\limits_{j \in 1 \ldots k} (r_{t(j)} - \bar{r_t})(t_{e(j)} - \bar{t_e})}{(k-2) \sum\limits_{j \in 1 \ldots k} (r_{t(j)} - \bar{r_t})^2}} \tag{3.5}$$

If the estimated standard error $se(b)$ is above a heuristically set confidence level C, we do not employ execution time prediction. Instead we report the last measured application execution time value rather than a predicted future one. In general however the last measured value is not an accurate predictor, as it ignores the current rate.

3.3.3 Rate Prediction

In this section we describe how we predict the rate $\hat{r}$ of an application, which we use to calculate the sum of the rates of all applications running on components of a node, $\hat{r_t}$. The latter is used to predict the application execution time $\hat{t_e}$ using Equation 3.4. We base the prediction of the rate of every application that is using a component hosted on the node on both auto- and cross-correlation. We take into account auto-correlation by building our prediction of a component's future input rate on its previous input rate. This captures any self-similarity the application traffic may have, which has been known to be the case for various types of traffic in stream processing environments [95]. We take into account cross-correlation, by also building our prediction of the input rate of a component on the current input rate of a previous component in the application component graph. This captures the fact that preceding components observe changes in the application input rate before the current component. Since data flow from one component to the next, the observed trends are often seen in the current component as well. In particular, we identify the preceding component m in the application component graph, the rate of which has the maximum correlation with the rate of the current component so far. In summary, we estimate the k-th input rate $\hat{r_k}$ of a component based on its previous input rate r_{k-1}, as well as the current and previous input rates of component m, $r_{k(m)}$ and $r_{k-1(m)}$ respectively.

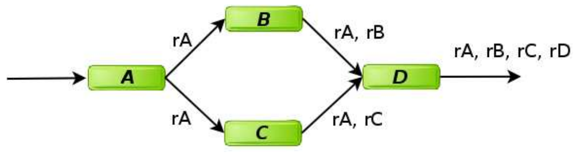


Figure 3.2: Propagation of rate values for correlated rate estimation.

We transfer the current input rate values to the downstream components using the same path followed by the data tuples, as shown in Figure 3.2. This way, for each of the previous i components in the application component graph, a series of $(k-1)$ pairs $(r, r_{(i)})$ is built. This series associates the $(k-1)$ rate values r of the current component with the $(k-1)$ rate values $r_{(i)}$ of each of the previous i components. We use the Pearson Product Moment R, a popular correlation coefficient [53], to estimate how the rate of each of the previous i components in the application component graph is correlated to the rate of the current component. We use the current (k-th) and previous ($(k-1)$-th) rates of the component m with the maximum correlation coefficient, $\arg_m maxR_{(k)}$ and $\arg_m maxR_{(k-1)}$ respectively, as predictors for the rate of the current component. Hence, assuming we have $(k-1)$ pairs of recorded input rates so far, the estimated input rate for the current component is:

$$\hat{r_k} = \frac{\arg_m maxR_{(k)}}{\arg_m maxR_{(k-1)}} \cdot r_{k-1} = \frac{r_{k(m)}}{r_{k-1(m)}} \cdot r_{k-1} \qquad (3.6)$$

and the component m is decided as the one with the maximum among all correlation coefficients R_i of each preceding component i in the application component graph:

$$R_i = \frac{\sum\limits_{j \in 1 \ldots (k-1)} (r_{j(i)} - \bar{r_{(i)}})(r_j - \bar{r})}{\sqrt{\sum\limits_{j \in 1 \ldots (k-1)} (r_{j(i)} - \bar{r_{(i)}})^2 \sum\limits_{j \in 1 \ldots (k-1)} (r_j - \bar{r})^2}} \tag{3.7}$$

where the average rate values of the i-th preceding and the current component, $\bar{r_{(i)}}$ and $\bar{r}$ respectively, are:

$$\bar{r_{(i)}} = \frac{\sum\limits_{j \in 1 \ldots (k-1)} r_{j(i)}}{k-1} \qquad \bar{r} = \frac{\sum\limits_{j \in 1 \ldots (k-1)} r_j}{k-1} \tag{3.8}$$

3.4 Application Hot-Spot Alleviation

3.4.1 Identifying the Components to Migrate

After an application hot-spot has been predicted, the next step is to determine which component execution(s) to migrate in order to resolve the hot-spot. We perform QoS projection and choose the migrations in such a way, so that the predicted execution times of the remaining applications in the node are within their QoS requirements.

Specifically, our goal is to determine the minimum number of migrations that will result to all the remaining applications satisfying their QoS requirements. In other words, we seek the minimum number of migrations that will reduce the sum of rates of all the applications in the node to such a degree, that all projected execution times for the remaining applications will be within their QoS requirements. More formally, and by building on the concepts introduced in Section 3.3, we migrate the component execution(s) that remove the minimum number of predicted rates $\hat{r}$ (from Equation 3.6), so that the predicted sum of application rates on the node $\hat{r_t}$ results to predicted execution times $\hat{t_e}$ (from Equation 3.4) such that, for every application remaining in the node, the slack time t_s (from Equation 3.1) is positive. This optimization problem lends itself to a dynamic programming solution in pseudo-polynomial time. After observing that usually one migration suffices to alleviate a hot-spot, and to minimize the execution time overhead, as migration decisions need to be taken online, we employ a simple heuristic of selecting for migration the component with the largest $\hat{r}$ until all slack times become positive.

3.4.2 Identifying the Target Nodes

Once a component the execution of which is to be migrated has been identified, the host to migrate to has to be decided. The choice for migration targets is made among the nodes that host the same component. Among them we try to identify a node probable to satisfy the migrating application's QoS requirements, while not violating the QoS of the applications currently running locally. Such nodes are most probable to be found among the ones that are predicted to be less loaded. Each node predicts its local load using linear regression, based on predicted rate values, using a methodology similar to the one described in Section 3.3.2. We use a simple model, according to which a component's load is proportional to the number of input data tuples it is receiving, which is an assumption also made by previous works [90, 95]. We store load information in a decentralized architecture on top of the DHT, as is described in the next Section (3.4.3).

3.4.3 Decentralized Resource Monitoring

DHTs have been successfully used for decentralized discovery, enabling the mapping of keys to nodes, and routing query messages in logarithmic time [74]. Our extension to the traditional DHT model in which one component offering an operator would be stored in the node responsible for the operator's key, involves one more level of abstraction, that of a distributed inverted index. Following this approach, the peer responsible for an operator's key stores in a repository handlers to several peers that host components that actually offer the operator [35], together with their loads. The handlers are basically identifiers of the peers, enabling the routing of messages to them through the overlay. This way we associate operator names with handlers to nodes hosting components offering these operators, together with the current load values of these nodes. For example, in Figure 3.4 peer B is responsible for keeping the handlers for the components that offer an aggregator operator and the loads of the hosting peers. Therefore it keeps the handlers and loads of peers B and C. While B happens to offer an aggregator component itself as well, this does not always have to be the case. While the peers have control over which components they host, they do not control which operator keys they are responsible for. This is determined by the DHT's protocol. Conversion of operator names to unique keys is done by applying a hash function (SHA-1 [55]). Configuration changes caused by node arrivals and departures are handled gracefully by the DHT. Whenever components are newly introduced or cease being offered, the peers hosting them register or unregister handlers for them on the DHT. The DHT also handles peer failures by transferring the responsibility for the operators for which a failed peer was responsible to other peers [74]. Finally, balancing the distribution of operator keys among nodes is also taken care of by the DHT protocol. Whenever a node's load changes, it consults the DHT to determine the nodes responsible for holding the handlers for all the components it offers. It then sends load update messages to them. For example, in Figure 3.4 node B that offers a filter, an aggregator, and a transcoder, will send its load update messages to the responsible nodes, C, B, and A, respectively.

Peers continuously monitor their load to enable load balancing decisions. The processor load is measured by parsing the `/proc` interface, where it is reported as the number of components in the run queue or waiting for disk I/O. Three load averages are maintained, namely the processor load during the last 1, 5, and 15 minutes.

Having each peer of the DHT maintaining handlers of all hosts offering a particular operator facilitates load monitoring. Specifically, each peer responsible for an operator is also responsible for monitoring the load of the peers that host components offering it. It stores the handlers to the hosts offering the components in a peer-handler repository, and their corresponding loads in a load repository. We note that load monitoring peers are also hosting components. Thus, all responsibilities are shared among all nodes in a true peer-to-peer fashion.

To avoid the communication overhead caused by polling, we enable the peers to inform the monitoring peers only when a significant change in their load occurs. Whenever a peer's load changes, it consults the DHT to determine the peers responsible for holding the handlers for all the components it offers. It then sends load update messages to them. For example, in Figure 3.4, peer B that offers a filter, an aggregator, and a transcoder, will send its load update messages to the responsible peers, C, B, and A respectively. One of

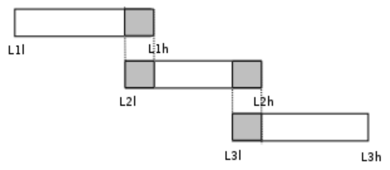


Figure 3.3: Example of three overlapping load levels.

the load update recipients happens to be peer B itself in this case. Propagating dynamic attributes, such as load measurements, efficiently, is a challenging task, as they may be need to be updated frequently. To reduce the communication overhead, our approach is to use k discrete levels of loads when describing the processing load of a peer. Each level is defined as an interval $L_i : [L_{il}, L_{ih})$, $L_{il} \leq L_{ih}$, within which the exact numeric load value is guaranteed to be. The basic idea is that the level eliminates all messages that are inside the interval. If the value of a load change l is small, *i.e.*, $l \in [L_{il} \leq L_{ih})$, then no messages need to be generated. Otherwise, load update messages need to be propagated. There is a tradeoff between the size of the interval and the number of messages generated. A narrow interval, *i.e.*, a small $|L_{ih} - L_{il}|$, enables more precise answers at the expense of a high communication cost, while a wide interval, *i.e.*, a large $|L_{ih} - L_{il}|$, reduces the communication cost but increases imprecision. To further reduce the number of messages propagated and also avoid cases where there is frequent change between two levels (*i.e.*, the load change is very small and falls in the boundaries of the levels, or there is load instability), we use overlapping levels; in which a small interval is common in two levels. Whenever the load changes but stays in the overlap region, no message needs to be generated. While load update messages are generated only when the load changes level, the actual load averages are then propagated instead of just the load level. Figure 3.3 graphically illustrates our overlapping levels approach. Load monitoring is performed by a different thread from the rest of the middleware functions such as stream processing. Thus, even if a peer has a high stream processing load, it can still send load update messages whenever needed. Load update messages can be propagated infrequently even if no load change occurred, to ensure that a peer has not failed. If a peer failure is detected, the components hosted by the failed peer need to be unregistered from the DHT and the applications that were using them need to recover using techniques such as the ones presented in [11].

3.4.4 Migration Protocol

By utilizing the load monitoring architecture a node determines the least loaded node offering the component the migration requires. After the migration target has been identified, the migration from the source to the target takes place, to resolve the application hot-spot. The execution of a particular component migrates to another peer that hosts the same component, redirecting the applications in which this component is participating. For example in Figure 3.4, the filter component of peer B is participating in two distributed applications, which will both be affected by the migration of it. While the component execution migrates, the component is not unregistered from the DHT, as the peer still has the capability of offering the same operator in the future.

To avoid QoS violations we perform QoS projection that predicts whether the QoS of the migrating and of the currently running applications will be able to be met after the migration has occurred. Once it has received a migration request, a node determines whether after accepting the migration it will be able to provide the migrating application its required QoS. Additionally, it determines whether the migration will not result to QoS violations for the locally executing applications. To achieve these goals, a migration target performs QoS projection involving the migrating and the currently running applications, that is similar to the one described in Section 3.4.1. Specifically, it ensures that by adding $\hat{r}$ for the new application, the sum of application rates on the node $\hat{r_t}$ will not result to a predicted $\hat{t_e}$ (from Equation 3.4) that results to a negative execution time slack for any application (from Equation 3.1). If that is the case, the migration is accepted and takes place using the migration protocol presented next. Our current migration mechanism caters to stateless components and simple components whose state is captured in small buffers. State transfer is a separate issue by itself and worth future investigation.

Our goal when choosing how to perform a migration is to offer no disruption in the application execution and minimal performance impact. While some stream applications in which data units arrive in very high rates may actually afford missing some data tuples during the transition, our migration mechanism also accommodates applications in which all data tuples must be delivered. The delivery of all data tuples in this case is checked using their sequence numbers, which indicate their requested delivery order. It is important to perform the migration in the least intrusive way, to avoid affecting the performance of the applications as much as possible. Therefore we strive to improve upon the existing approaches of pausing the application execution or buffering incoming data tuples to be processed later [80, 96]. Thus, the connections from the new component to the upstream and downstream components are updated offline.

In more detail, our non-disruptive migration protocol is as follows:

Phase 1. Once a source node decides the migration of an application to a target, it sends a *migration request* message to the target, in which the information for the corresponding component that needs to be invoked is included. This includes which component's code will be executing, as well as which upstream and downstream components it will be communicating with. Finally, information regarding the QoS of the migrating application is also included in the request, specifically, the application's predicted rate $\hat{r}$, as well as the data for Equations 3.1 and 3.4, including the (t_e, r_t) series. In Figure 3.4, B sends a migration request to A, requesting from it to take over the filter execution for the application its filter component was currently participating.

Phase 2. Once it has received a migration request, the target has to determine whether after accepting the migration it will be able to provide the migrating application its required QoS. Additionally, it has to determine whether the migration will not result to QoS violations for the locally executing applications. To achieve these goals, the target performs a QoS projection involving the migrating and the currently running applications, that is similar to the one described in Section 3.4.1. Specifically, it ensures that by adding $\hat{r}$ for the new application, the sum of application rates on the node $\hat{r_t}$ will not result to a predicted $\hat{t_e}$ (from Equation 3.4) that results to a negative execution time slack for any application (from Equation 3.1). If that is the case, the migration is accepted, and the target instantiates the component with the parameters that were included in the migration request.

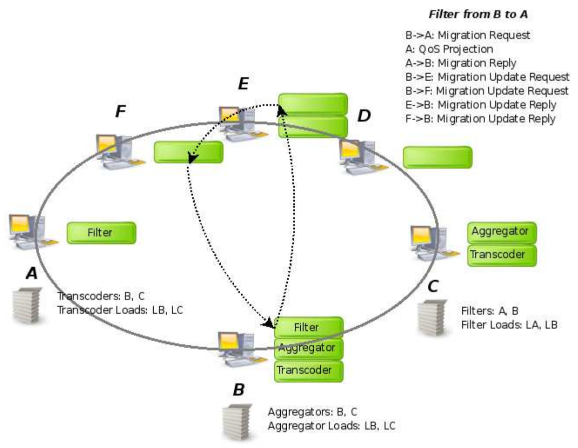


Figure 3.4: Migration example.

Phase 3. If the migration can be accepted, once the new component has been instantiated, the target sends a *positive migration reply* message to the source, stating that it is ready to accept data tuples. In Figure 3.4, the migration reply is sent from A to B. If QoS violations were projected, a *negative migration reply* is sent to the source. In this case, the source tries the next least loaded node offering the required component, until it receives a positive migration reply. If all the nodes offering the specific component reply negatively, a new component needs to be instantiated to resolve the hot-spot, according to the current policy for placing components in the network, which is outside the scope of this work [4, 61].

Phase 4. After receiving a positive migration reply, the source sends *migration update request* messages to all upstream components of the component participating in the migrating application, ordering them to update their downstream component to point to the one residing on the target. In Figure 3.4, a migration update request is sent from B to F.

Phase 5. The upstream components reply with a *migration update reply* message each, specifying that they have updated their downstream components. In addition they specify the sequence number of the last data tuple they sent before the update. They now send any further data tuples to the new component. In Figure 3.4, a migration update reply is sent from F to B.

Phase 6. Once the source has collected replies from all upstream components and has processed the data tuples they specified as the last ones to be sent to it, it knows that the migration has completed. It then kills the application which has migrated.

Paying attention to the sequence numbers allows our protocol to know when a migration can complete without disrupting the application execution. Updating the connections from the upstream components independently enables our protocol not to affect the application execution.

A common problem of migration protocols is the ping-pong phenomenon, manifesting itself as migrating processes or threads bouncing between two nodes. Such ill behavior is avoided in our case due to our QoS projection, which enables us to predict potential QoS violations and avoid poor migration choices.

3.5 Experimental Evaluation

To evaluate the performance of our hot-spot prediction and alleviation mechanisms we have implemented them in our Synergy distributed stream processing middleware and performed experiments over the PlanetLab [15] wide-area network testbed. We used 34 hosts, each one of them issuing a request for a distributed stream processing application.

Each node was hosting stream processing components that were processing data tuples as they arrived. We set the application end-to-end delay QoS requirement to 20s.

To evaluate the accuracy of our prediction mechanisms we implemented a real stream processing application from the network traffic management domain, which we fed with real TCP traffic traces. We used a stream processing application from the Stream Query Repository [83], in which, assuming a packet capturing device installed in a network, a system administrator wishes to monitor the source-destination pairs in the top 5 percentile in terms of total traffic in the past 20 minutes over a backbone link. We generated the streaming data to be processed by replaying a TCP traffic trace available from the Internet Traffic Archive [87]. Similar results where obtained with the rest of the traces from [87]. The trace contained two hours' worth of all wide-area TCP traffic between the Lawrence Berkeley Laboratory and the rest of the world, consisting of 1.8 million packets. Each packet contained a timestamp, and fields defining the source and destination (IPs and ports), as well as the size of the packets exchanged between them, as shown in Figure 3.5. Our implementation of the above stream processing application to process the packet input over 20-minute windows to generate the monitoring output involved eight components[1]. Each node instantiated a different stream processing application that included all eight components of the application component graph, distributed randomly on different nodes of the system. Each node predicted the rate and the execution time of the components it was hosting using the statistical methods described in Section 3.3. We plot predicted and actual values to show correlation and burstiness.

[1]Application screenshots available at http://synergy.cs.ucr.edu/screenshots.html

timestamp	sourceIP	destinationIP	sourcePort	destinationPort	size

Figure 3.5: Format of the processed TCP packets.

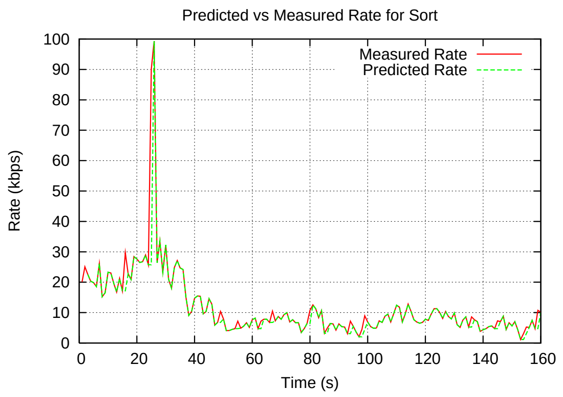


Figure 3.6: Rate prediction accuracy for "sort".

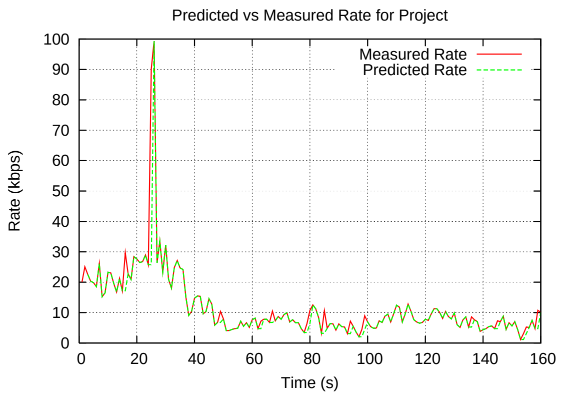


Figure 3.7: Rate prediction accuracy for "project".

Rate Prediction Accuracy. In our first set of experiments we investigated the accuracy of our rate prediction algorithm described in Section 3.3.3. Figures 3.6, 3.7, 3.8, 3.9, and 3.10 compare the predicted rate for the individual components of an application to their actual rate. Similar results were obtained for all applications, as well as for the rest of the components of the application component graph. We observe that the predicted rate closely follows the measured rate for the different component types, namely sort, project, aggregate, count, and compare. Another interesting observation is the correlation in the rate between different components, for example between sort and project, or between aggregate and count. This indicates the significance of cross-correlation between different components in the application component graph, which we exploit in addition to auto-correlation to predict component rates.

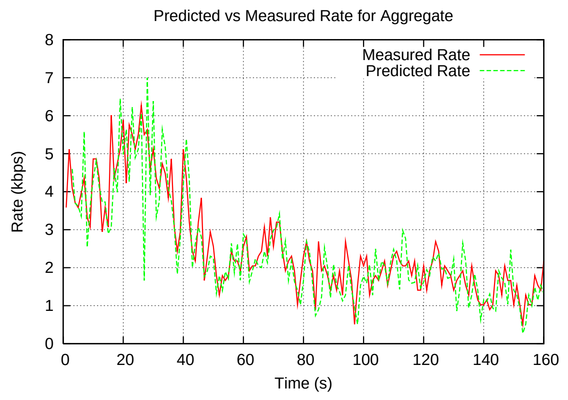


Figure 3.8: Rate prediction accuracy for "aggregate".

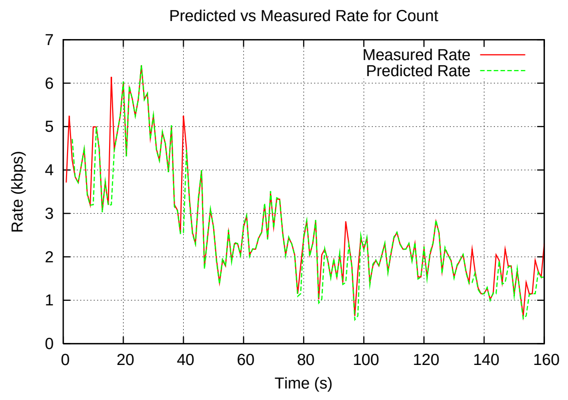


Figure 3.9: Rate prediction accuracy for "count".

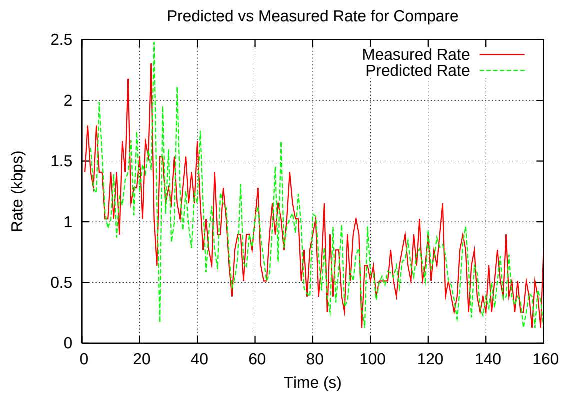


Figure 3.10: Rate prediction accuracy for "compare".

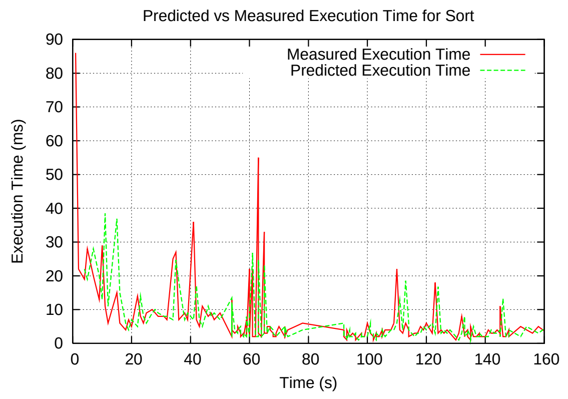


Figure 3.11: "Sort" execution time prediction accuracy.

Execution Time Prediction Accuracy. In our second set of experiments we investigated the accuracy of our execution time prediction algorithm described in Section 3.3.2. Figures 3.11, 3.12, 3.13, 3.14, and 3.15 compare the predicted to the actual execution time for the same set of components as in the rate prediction accuracy experiment. As was described in Section 3.3.2, the predictions are based on the sum of rates being processed by the node hosting each component. Note, that each component was hosted on a different node. The predicted execution time follows the execution time we measure. Cases

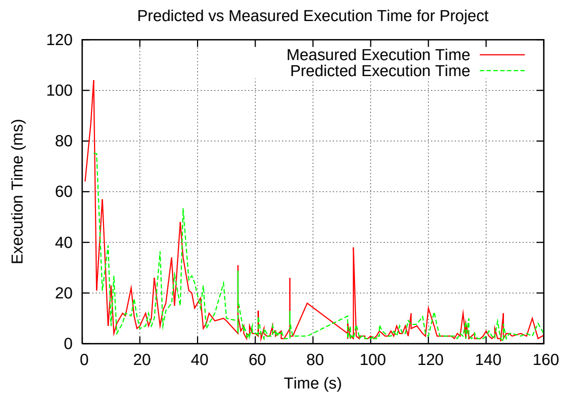


Figure 3.12: "Project" execution time prediction accuracy.

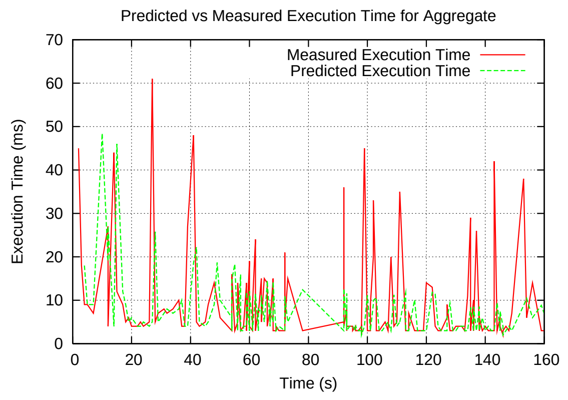


Figure 3.13: "Aggregate" execution time prediction accuracy.

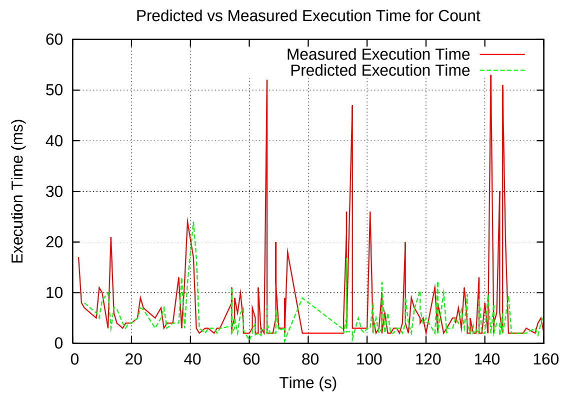


Figure 3.14: "Count" execution time prediction accuracy.

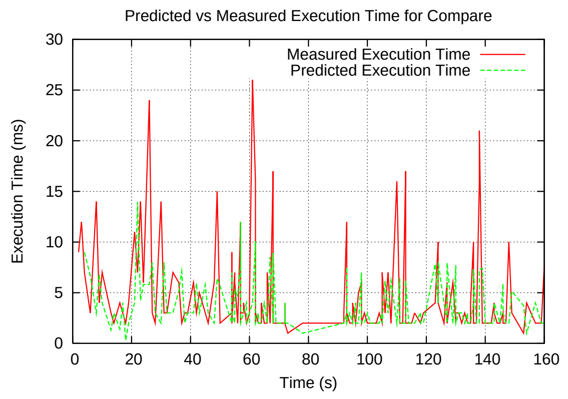


Figure 3.15: "Compare" execution time prediction accuracy.

where the prediction is very inaccurate are detected using the estimated standard error of the linear regression, as was described in Section 3.3.2. This way, instead of the inaccurate predicted future execution time value the currently monitored value is reported.

Execution Time Distribution. In our third set of experiments we investigated the relationship between the execution time of the individual application components and the total rate for all applications being processed by each node hosting a component, shown in Figures 3.16, and 3.17 (similar Figures were obtained for the rest of the components).

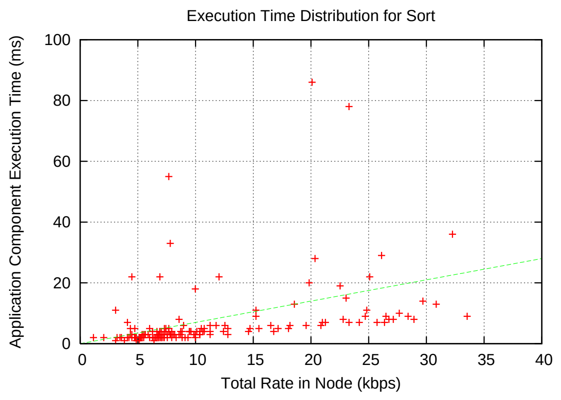


Figure 3.16: Execution time distribution for "sort".

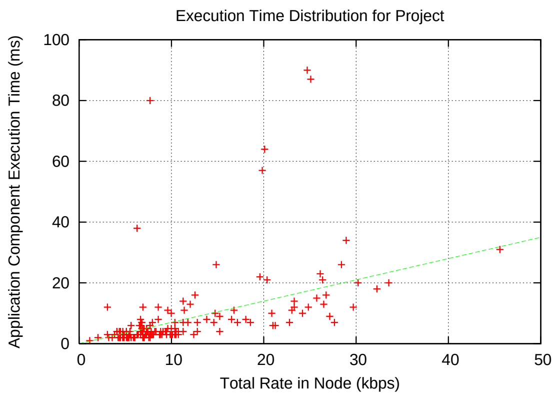


Figure 3.17: Execution time distribution for "project".

This enabled us to determine the accuracy of assuming a linear relationship between the two, which formed the basis of our linear regression-based execution time prediction algorithm described in Section 3.3.2. We observe that the relationship can be approximated by a line, excluding a few outliers. However this linear relationship is most evident when the total rate in the node is significant. If the node is lightly loaded, no significant queueing delays occur and therefore no significant variations in the execution time take place.

Prediction Parameters. In our fourth set of experiments we investigated various parameters regarding the prediction overhead and performance. In Figure 3.18 we show how rate prediction accuracy is affected when reducing the prediction frequency. Reducing the prediction frequency can enable the system to handle high rates, by avoiding the prediction overhead for every data tuple. We present the effect on prediction accuracy for the different components, when predicting the rate for every 1, every 50, every 75, and every 100 incoming data tuples. We observe that even by reducing the prediction overhead by a factor of 75,

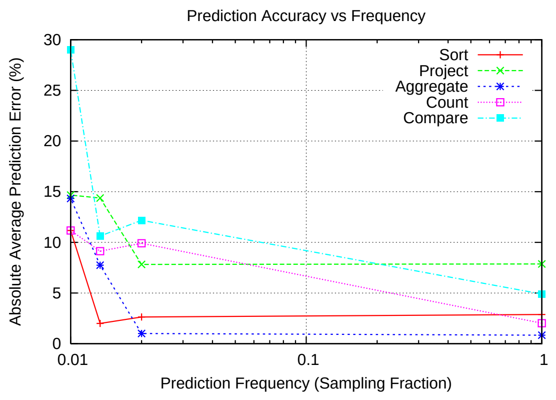


Figure 3.18: Rate prediction accuracy versus prediction frequency.

Component	Avg. Pred. Error (%)
sort	2.875
project	7.872
aggregate	0.838
count	2.019
compare	4.904

Figure 3.19: Absolute average rate prediction error.

Component	Avg. Pred. Time (ms)
sort	0.133
project	0.327
aggregate	0.509
count	0.836
compare	1.187

Figure 3.20: Average total prediction time.

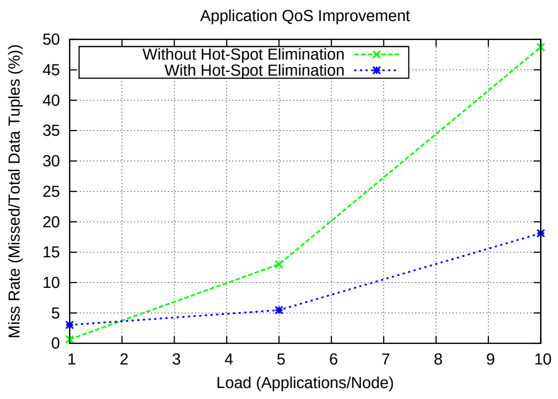


Figure 3.21: Application QoS improvement.

the prediction accuracy only drops by 8.775% on average, ranging from 2.0% for sort, to 14.375% for project.

Table 3.19 shows the average rate prediction error for the different application components. This provides a clear overview of the prediction accuracy. Even though some variation depending on the component semantics exists, the average prediction error is kept at 3.7016%. Table 3.20 shows the overhead in processing time for rate, execution time, and load prediction. The average overhead is 0.5984ms, which makes our algorithms suitable for online prediction.

Application Performance. In our fifth set of experiments we investigated the application benefits gained from our hot-spot prediction and alleviation mechanisms. Figure 3.21 shows the improvement in application QoS achieved by predicting application hot-spots and alleviating them using migration. The QoS metric displayed is the miss rate, defined as the number of data tuples that missed their QoS deadline, over the total number of data tuples that were produced by the source. The miss rate is displayed as a function of the system load. We inject additional load in the system by increasing the number of application component graphs each node requests from 1 to 10. When the system is underloaded not many application hot-spots occur and therefore their alleviation does not offer significant QoS advantages. However, as the system load increases, the miss rate increases drastically when hot-spots are not handled. Application hot-spot elimination controls this increase.

Figure 3.22 shows the benefit of hot-spot prediction and alleviation for the application performance. The performance metric displayed is the end-to-end application delay. Note that this delay is calculated only for the data tuples that did not miss their deadlines, as the ones that missed their deadlines are dropped by the local schedulers before reaching the receiver. While hot-spot prediction and alleviation enables the delivery of more data tuples as the load increases, it also maintains a lower average application end-to-end delay.

Figure 3.23 shows how a migration affects the performance of a particular application. For a load of 10 application requests per node, we show the end-to-end delay

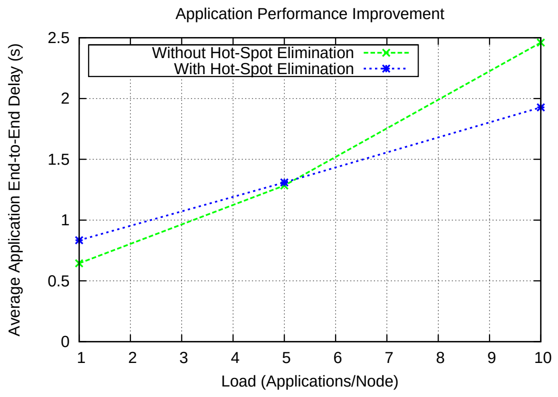


Figure 3.22: Application performance improvement.

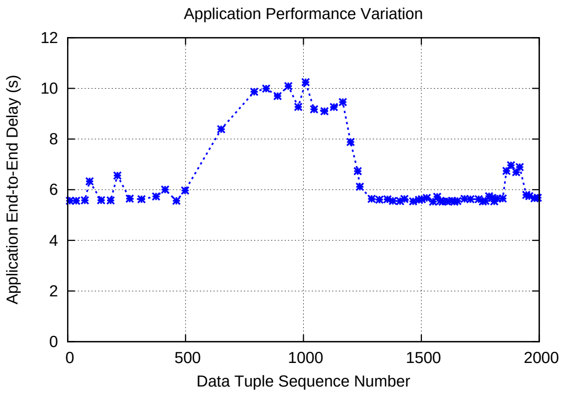


Figure 3.23: Application performance variation.

attained by delivered data tuples of one application. Approximately at data tuple #500 an application hot-spot occurs, resulting to an increase in the end-to-end delay. Our hot-spot elimination mechanism kicks in and decreases the end-to-end delay through migration approximately at data tuple #1200. It is also important to note that only the data tuples that were delivered within the application's QoS requirements are shown. As the application end-to-end delay increases, we can clearly observe a reduction in the number of delivered data tuples. After the hot-spot has been eliminated, the number of data tuples that miss their deadline decreases again and more points can be seen in the graph. The Figure magnifies, focusing on 2000 of the total tuples.

Load Management Overhead. In our sixth set of experiments we investigated the overhead associated with migration and load monitoring. In Figure 3.24 we show the migration overhead to achieve the hot-spot alleviation benefits. The number of migrations is shown as a function of the number of applications deployed in the system. We observe that the number of migrations grows linearly to the number of applications. On average, one

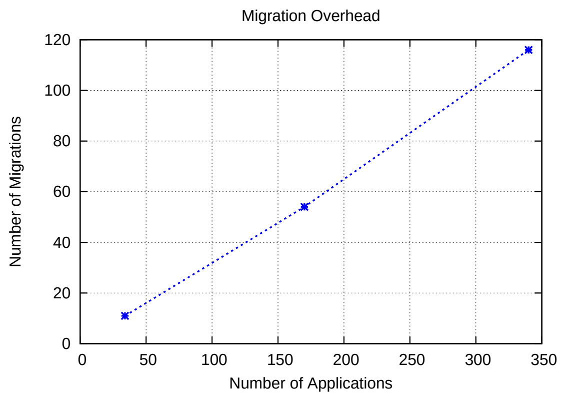


Figure 3.24: Migration overhead.

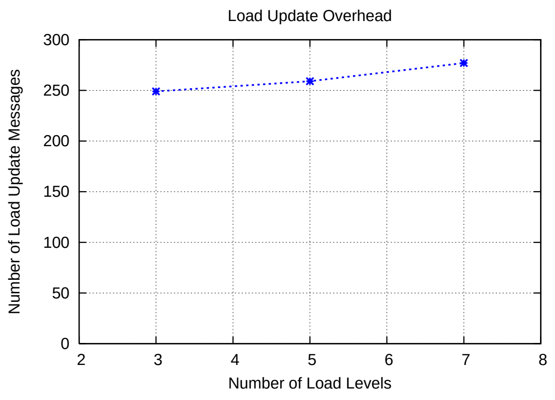


Figure 3.25: Load update overhead.

migration every three applications is required. This shows that on average one every three applications experience a hot-spot at some point during the execution, which motivates the need for application-oriented hot-spot alleviation. This assumes that not many applications require more than one migration, in other words that the system is not so overloaded that a migration does not permanently resolve a hot-spot. We also measured the average time required to perform a migration to be 1144ms. This time included the complete distributed protocol execution described in Section 3.4.2. The short migration time, together with the fact that our migration protocol enables application execution to continue while the migration is taking place offline, make our hot-spot alleviation mechanism suitable for distributed stream processing applications with QoS demands. Prediction further facilitates fast reaction to a hot-spot, before massive QoS violations occur.

Finally, to drive the migration decisions, load updates are generated as was discussed in Section 3.4.3. The overhead of the load monitoring mechanism is shown in Figure 3.25, as the number of load update messages propagated for different numbers of load levels. We always attribute the highest load level to any load above 10.0, which corresponds to a severe overload, and assign the rest of the load levels to loads from 0 to 10.0 accordingly. By varying the number of load levels from 3 to 7 we observed a moderate increase in the load update messages from 249 to 277. Thus, increasing the accuracy of our load monitoring mechanism only incurs a moderate increase in the load update overhead.

3.6 Conclusions

In this chapter we have described hot-spot prediction and alleviation mechanisms for distributed stream processing applications. Synergy's algorithms for hot-spot prediction are based on the statistical methods of linear regression and correlation, utilizing only light-weight, passive measurements. Statistics collection and hot-spot prediction and alleviation are carried out at run-time by all nodes independently, building upon a fully decentralized architecture. To alleviate hot-spots we empower nodes to autonomously migrate the execution of stream processing components using a migration protocol that offers minimal disruption in the application execution. The experimental evaluation of our techniques on the Synergy middleware over PlanetLab, and using a real network monitoring application operating on traces of real TCP traffic, demonstrated high prediction accuracy and substantial performance benefits with moderate monitoring and migration overheads.

Chapter 4

Distributed Replica Placement for High Availability

4.1 Introduction

In the previous chapters we focused on the end-to-end delay QoS metric. In this chapter we focus on availability as our QoS metric and look at the problem of high availability in a distributed stream processing system. By taking into account the particular characteristics of stream processing applications we first identify design principles for a replica placement algorithm for high availability. We incorporate these principles in a decentralized replica placement protocol that aims to maximize availability, while respecting resource constraints, and making performance-aware placement decisions. We integrate our replica placement protocol in Synergy, our distributed stream processing middleware and compare its performance with the current state of the art.

The rest of this chapter is organized as follows: We begin by providing a description of the system model and notation we use in our discussion in Section 4.2. We then present the design principles behind our replica placement algorithm in Section 4.3, followed by the protocol that implements them in Section 4.4. Section 4.5 presents our experimental evaluation, before we conclude the chapter in Section 4.6.

4.2 System Model

We begin by defining our system model. Table 4.1 summarizes our notation. Each Synergy node v_i is characterized by its current processor load p_{v_i} and its residual processing capacity rp_{v_i}, which are inferred from the CPU idle time as measured from the `/proc` interface. The residual available bandwidth rb_{e_j} on a virtual link e_j between v_i and a remote node is calculated using a bandwidth measuring tool (e.g., Iperf [56]). We also use b_{e_j} to denote the amount of current bandwidth consumed on e_j. Finally, the communication latency of a virtual link e_j between v_i and a remote node is measured using direct pings, even though more elaborate latency calculation methods can be integrated [92].

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 $\sum is_i$ and produces a set of output streams $\sum os_i$. Stream processing components can have more than one inputs (e.g., a join operator) and outputs (e.g., a split operator). Each atomic operator can be provided by multiple component instances $c_1, \ldots, c_k$, which we call *component replicas.*

Notation	**Meaning**
ξ	Query Plan
λ	Application Component Graph
ρ	Replication Component Graph
ϱ	Replication Degree
n	Number of Components in Application
k	Number of Components on a Node
A	Availability of Application
F	Failure Probability of Application
a_i	Availability of Component
f_i	Failure Probability of Component
α	Availability of Node
ϕ	Failure Probability of Node
v_i	Node
e_j	Virtual Network Link
o_i	Operator
s_j	Stream
c_i	Component
l_{e_j}	Latency of Virtual Link e_j
p_{o_i}	Processing Time Required for Operator o_i
b_{s_j}	Bandwidth Required for Stream s_j
p_{v_i}	Processor Load on Node v_i
b_{e_j}	Network Load on Virtual Link e_j
rp_{v_i}	Residual Processing Capacity on Node v_i
rb_{e_j}	Residual Network Bandwidth on Virtual Link e_j

Figure 4.1: Notations.

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. Figure 1.1 shows an example of a query plan. The CPU processing time 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. Bandwidth requirements are calculated according to the user-requested stream rate, while processing time requirements are calculated according to the data rate and resource profiling results for the operators [68]. Processing and bandwidth requirements can represent average or worst-case load, depending on the robustness required from the application instantiation.

The query plan is dynamically instantiated into an *application component graph*, denoted by λ, depending on the particular components that are being used by the application. 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_k if the output of component c_i is the input for component c_k.

We assume a primary/backup, passive replication scheme [17,27]. Each component has a primary and a number of backup replicas. The primary component replicas are the vertices of the application component graph. With each stream s_j flowing between primary components c_i and c_k we associate a required bandwidth b_{s_j}, and with the corresponding virtual network link e_j we associate a latency l_{e_j}. For each of the primary component replicas there exist one or more *backup* component replicas. The backup component replicas are passive replicas in the sense that they do not process data streams, but they asynchronously replicate the output of the primary replicas to be able to take over in case their primary counterparts fail. This enables faster recovery compared to instantiating components after a failure occurs. State transfer between the primary and backup replicas is not the focus of this work and existing solutions for consistency, checkpointing, failure masking, and recovery for distributed stream processing systems [11, 38, 79], as well as solutions based on view-synchronous communication [16], can be integrated in our architecture. Let b_{ix} be the bandwidth needed to do a state transfer of a primary component c_i to its xth backup replica and let l_{ix} be the communication latency of the corresponding virtual network link between the two replicas. Since in our implementation the state transferred from a primary replica to its backup replicas is the primary's output, essentially $b_{ix} = b_{s_j}$ for all of a component's backup replicas. The primary and backup component replicas are the vertices of a disconnected, directed graph, called the *replication component graph*, denoted by ρ. The edges of this graph represent the replication of the output of the primary component replicas to their backup counterparts.

Component replicas are hosted by different nodes (machines) in the system. We call the number of replicas of a component the component's *replication degree* ϱ. ϱ is defined in an application request. We denote with n the number of primary component replicas needed by a composite distributed stream processing application, which corresponds to the number of vertices in the application component graph. We denote with k the number of component replicas belonging to a particular application that are being hosted by a single node. Essentially, k represents the number of component replicas of an application that are collocated in a single node.

A node is available with probability α, or fails with probability $(1 - \alpha)$, which we define as its failure probability ϕ. ϕ includes both the failure probability of the node itself and of the network links connecting it to other nodes. Not having any historical failure data, we assume that all nodes fail with the same probability ϕ. We only consider independent, fail-stop failures, and once a node has failed we regard all the components hosted by it, and all the applications using these components, as permanently failed. We assume a reliable communication protocol such as TCP, and that network partitions are handled by redundancy in the routing tables of the DHT substrate, or by some other mechanism that guarantees eventual consistency [11,49]. We define the availability a_i of a component c_i, as the probability that at least one of its replicas is available (executing correctly and reachable over the network). The probability $(1 - a_i)$ we define as unavailability or failure probability f_i of a component c_i.

When an application request arrives, our goal is to place component replicas in a way that maximizes the availability of the application to be instantiated. For placement decisions that are equivalent in terms of availability we also seek to maximize the application's performance. We define the availability A of a composite application as the probability that all its components are available (executing correctly and reachable over the network). The probability $(1 - A)$ we call unavailability or failure probability F of the composite application. In other words, we seek to maximize the percentage of successful requests for composite applications.

4.3 Designing Replica Placement for High Availability

We now describe the design principles of our high availability placement algorithm, focusing on the three aspects introduced in Section 4.1, namely: i) Determining a placement of component replicas that maximizes availability (Section 4.3.1). ii) Determining the number of nodes to use for placing the component replicas according to the system's resource availability (Section 4.3.2). iii) Determining where to place the component replicas across nodes to maximize application performance (Section 4.3.3).

4.3.1 Maximizing Application Availability

We first look at the problem of determining a placement of component replicas that maximizes the availability of a distributed stream processing application. We look at the two important characteristics of distributed stream processing applications, namely the fact that they are composite and strict, to understand their availability requirements that differentiate them from other distributed applications, and to guide our placement decisions. In many distributed applications such as distributed storage [3, 5, 42, 75] or client/server computing [14, 27, 31, 45, 51, 64, 93] that focus on the availability of individual objects, such as files or processes, an increase in the number of replicas usually implies a similar increase in the availability of the application. While we expect that increasing the number of component replicas will also increase the availability of a distributed stream processing application composed of them, we find that this increase greatly depends on the relative placement of the individual component replicas. Moreover, unlike applications that can tolerate missing objects, such as ones based on majority voting or erasure coding, a distributed stream processing application requires all of its components for correct execution. In short, how the individual component replicas are placed on nodes affects whether all components will be available for the application execution.

To gain a better understanding of how component placement affects application availability we conduct a simple simulation. We place components with replication degree

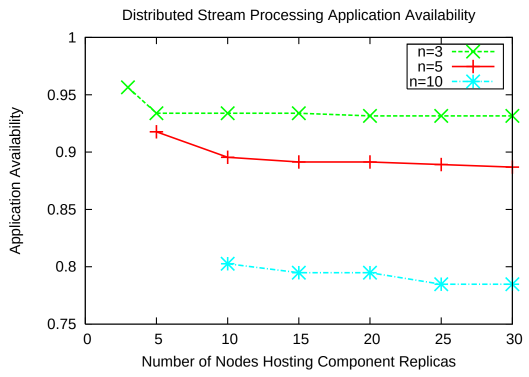


Figure 4.2: Availability decreases with larger application component graphs and increases as components are concentrated in fewer nodes.

$\varrho = 2$ on a subset of 30 nodes. We then calculate using recursion all possible combinations of 5 failed nodes and determine the average application availability from all failure combinations. We simulate applications with $n = 3, 5, 10$ components. Figure 4.2 summarizes the results and helps us reach the following conclusions: The relative placement of the individual components affects the availability of the composite application. More specifically, concentrating the component replicas in a smaller than $\varrho \cdot n$ subset of nodes increases application availability. The reason lies in the fact that the composite application is strict. Please note that spreading the components across more nodes would have the opposite effect for an application where even a subset instead of all the components being available would suffice. Finally, we observe that as the size n of the application component graph increases, application availability decreases. This is because more nodes need to be employed for hosting all component replicas (since no replicas of the same component can be hosted by the same node), while all of the components need to be available for the application to be available. Similar conclusions are drawn with larger replication degrees as well.

From the above discussion we reach the conclusion that collocating as many component replicas as possible in the smallest number of nodes, in other words maximizing the number k of components hosted by a node, maximizes the availability of a distributed stream processing application. The larger the k the larger the availability A achieved. Thus, taking into account that replicas of the same component should be placed on different nodes, an optimal replica placement algorithm for a distributed stream processing application would place all primary component replicas on a node, and use another $\varrho - 1$ nodes to place the backup component replicas, placing n backup replicas on each node. In practice however such a placement is infeasible due to the distributed nature of the applications and also the resource constraints imposed by the nodes. Yet, we show in the experimental evaluation in Section 4.5 that the availability achieved by our replica placement algorithm, that takes into account the system's resource constraints, is comparable to that of this optimal placement.

Existing research efforts on component placement for distributed stream processing systems [4, 61] place components to nodes with performance optimization in mind and ignore how components are placed relative to each other. This results to a random relative component placement. However, [98], which is the only study of the availability of *multi-object* operations in distributed systems we are aware of, has shown that random placement offers the worst availability for multi-object operations that cannot tolerate missing objects. Furthermore, [98] has shown that for multi-object operations that cannot tolerate missing objects the highest availability is provided by increasing inter-object correlation. In our work we maximize inter-object correlation by placing all replicas in the smallest possible group of nodes, as long as the nodes' resources allow us to do so. We show in the experimental evaluation in Section 4.5 that placing components ad-hoc, per application request, allows us to achieve higher availability than a placement algorithm that partitions nodes to groups and statically pre-assigns replicas to these groups, which is the one that performs best for strict multi-object operations in [98]. The fact that the application performance also needs to be taken into account in placement decisions, as it is affected by inter-component communication, further differentiates our replica placement for distributed stream processing applications from replica placement for other distributed applications such as those presented in [98] (e.g., storage).

4.3.2 Respecting Resource Availability

While our investigation so far suggests that application availability would be maximized by placing all n components of an application component graph on a node, we now discuss why such a placement is infeasible in practice and identify the resource constraints that determine a number of replicas per node $k \leq n$. The two resource constraints that affect component replica placement in practice are processing capacity and network bandwidth. To host a primary component replica, a node needs processing capacity to process its input stream(s), downstream bandwidth to receive its input stream(s), upstream bandwidth to transfer its output stream(s) to the next primary component replicas in the application component graph, and upstream bandwidth to transfer its output stream(s) to its backup component replicas. To host a backup component replica, a node needs downstream bandwidth to receive its input stream(s).

As described in Section 4.2, the monitoring module of a node v_i collects information regarding its residual processing capacity rp_{v_i} and the residual network bandwidth rb_{e_j} on each virtual link e_j between v_i and another node. The bandwidth and the processing time requirements of a component are included in the query plan ξ of an application request. Bandwidth requirements b_{s_j} are calculated according to the user-requested stream rate, while processing time requirements p_{o_i} are calculated according to the data rate and resource profiling results for the operators, as described in Section 4.2.

Thus, to be able to host a primary component replica c_i, node v_i needs: $p_{o_i} \leq rp_{v_i}$, $b_{s_j} \leq rb_{e_j}, \forall s_j \in \sum is_i$, and $b_{s_j} \leq rb_{e_j}, \forall s_j \in \sum os_i$, for all virtual links e_j connecting primary to primary and primary to backup component replicas. To be able to host a backup component replica c_i, node v_i needs: $b_{s_j} \leq rb_{e_j}, \forall s_j \in \sum os_i$, for the virtual link e_j connecting the backup replica to its primary counterpart. For example, in Figure 4.4, node v_{21} can host primary replica c_{21} only if $p_{o_2} \leq rp_{v_{21}}$, $b_{s_4} \leq rb_{e_{21\_41}}$, and $b_{s_4} \leq rb_{e_{21\_22}}$. Node v_{12} can host backup replica c_{12} only if $b_{s2} + b_{s3} \leq rb_{e_{11\_12}}$.

Thus, we collocate replicas on nodes as much as their resources permit it. Therefore, the nodes' processing capacity and the virtual links' network bandwidth ultimately determine the minimum number of nodes that can be used for placing replicas in practice.

4.3.3 Maximizing Application Performance

While application availability is only affected by the number of nodes that are used for placing component replicas, application performance is also affected by the particular nodes used for placement. Moreover, both availability and performance are affected by the relative placement of the component replicas among nodes. For placement decisions that are equivalent in terms of availability we seek to maximize the application's performance. To determine which nodes to use for placing the component replicas and where to place the component replicas across these nodes to maximize application performance we look at the two types of communication that affect it: i) *Inter-operator communication*: The primary component replicas that participate in a distributed stream processing application exchange data in the form of input and output streams. ii) *Intra-operator communication*: The primary component replicas asynchronously replicate their output streams to their backup component replicas. In the application component graph example of Figure 4.3 component replicas c_{11} and c_{12} offer operator o_1, c_{21} and c_{22} offer o_2, c_{31} and c_{32} offer o_3, and c_{41} and c_{42} offer o_4. Thus, as Figure 4.4 shows, inter-operator communication takes place between c_{11}, c_{21}, c_{31}, and c_{41}, while intra-operator communication takes place between c_{11} and c_{12}, c_{21} and c_{22}, c_{31} and c_{32}, and c_{41} and c_{42}.

To capture the two aforementioned types of communication we define the two corresponding communication costs for the entire application component graph: i) The **inter-operator communication cost** is defined as $c_{inter} = \sum\limits_{j \in 1 \ldots n} s_j \cdot l_{e_j}$ and captures the cost of transferring the streaming data through the primary replicas of the application component graph, with bandwidth requirements s_j and link latencies l_{e_j}. ii) The **intra-operator communication cost** c_{intra} is defined as $c_{intra} = \sum\limits_{i \in 1 \ldots n} \sum\limits_{x \in 1 \ldots \varrho} s_{ix} \cdot l_{ix}$ and captures the cost of transferring the output of the primary replicas to their backups, with bandwidth requirements s_{ix} and link latencies l_{ix}.

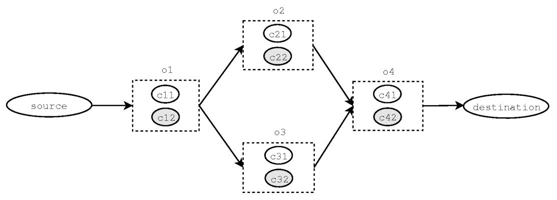


Figure 4.3: A simple distributed stream processing application.

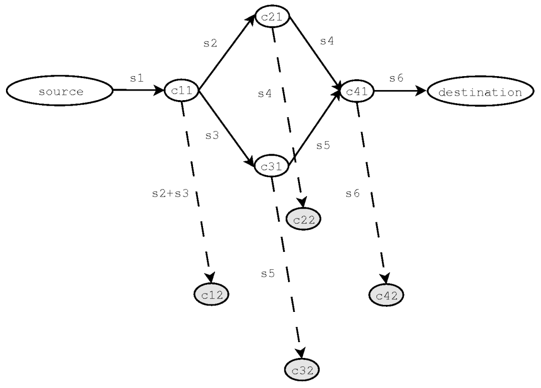


Figure 4.4: Replication in a simple distributed stream processing application.

Thus, the component placement problem is defined as a constrained optimization problem, where the goal is to determine the smallest group of nodes to host the $\varrho \cdot n$ component replicas of an application, that minimizes the total communication cost $c_{inter} + c_{intra}$, such that no replicas of the same component are hosted by the same node, and the processing and bandwidth constraints are met.

Previous work on component placement to improve application performance [4,61] has considered the simpler version of the problem without replicated components. In this case, the placement problem is reduced to the placement of only the primary component replicas, in other words the construction of the application component graph. Even in this simpler case, finding an optimal solution is an NP-complete problem [4].

Since an optimal solution is not available, we propose a greedy one that: i) Places the primary component replicas so that the inter-operator communication cost is minimized. ii) Places the backup component replicas so that the intra-operator communication cost is minimized. We use virtual link latencies to guide the placement decisions of primary and backup component replicas, to minimize the inter- and intra-operator communication costs respectively. While the amount of data s_j that needs to be transferred in every case is defined by the application and cannot be affected by the placement decisions, we take it into account in the placement decisions by weighing link latencies with s_j. The fact that we try to minimize the number of nodes to use for component replica placement, to maximize availability, limits the number of latency measurements we need to perform and simplifies the replica placement problem.

4.4 Distributed Placement Protocol

We now present our protocol for placing the component replicas of a distributed stream processing application. Our primary goal is to provide a scalable distributed protocol that determines replica placement for high availability of the application. For placement decisions that are equivalent in terms of availability, we seek to maximize the application's performance. Our protocol implements in a decentralized manner the three decisions regarding: i) The collocation of components to maximize application availability (Section 4.3.1), ii) the number of nodes to use for placement so that system resources are not exceeded (Section 4.3.2), and iii) which nodes to use so that the communication costs are minimized (Section 4.3.3). Component collocation is achieved by reusing nodes for placement as much as possible. Resource overloads are avoided by taking into account the nodes' processing capacity and the virtual links' network bandwidth in the placement protocol. Finally, inter- and intra-operator communication costs are minimized by taking into account the virtual links' latencies in the placement decisions.

The placement algorithm takes as input a user stream processing request, described by a query plan ξ and the component's replication degree ϱ. The output of the placement algorithm is an application component graph λ, specifying the primary component replicas that accomplish the application execution, and the nodes that are hosting them, as well as a replication component graph ρ, specifying the backup component replicas that replicate the output of the primaries, and the nodes that are hosting them.

Component replica placement decisions are carried out hop-by-hop. To this end, we have implemented the following types of messages: Placement requests, placement replies, placement negotiations, and placement decisions. We describe the contents of these messages as we introduce their role in the placement protocol. We now present the details of the placement protocol, which include its three phases, namely the bootstrapping, the propagation, and the completion, and focus on the six steps for primary and backup placement, that are executed on every hop. A high level description of the placement algorithm is shown in Figure 4.5.

Phase 1. Bootstrapping. The protocol execution begins with the submission of a user request for a stream processing application, described by a query plan ξ, and the replication degree ϱ of the application's components. A user request is submitted directly to a node v_s, if the client is running the middleware, or redirected to a node v_s that is closest to the client based on a predefined proximity metric (e.g., geographical location). v_s bootstraps the placement protocol by sending the user request to the node(s) that host the inputs of the application, which we call the *source nodes.* These nodes are usually pinned

```
Input:   query plan $\xi$, replication degree $\varrho$, node $v_s$
Output:  application component graph $\lambda$,
         replication component graph $\rho$
for each node $v_i$ in path
    perform transient resource allocation at $v_i$
    identify candidate nodes already used for placement
    select candidate nodes meeting bandwidth requirements
    sort candidate nodes by latency
    for each primary replica of downstream component
        send placement request or placement negotiation
        receive placement reply
        send placement decision
    for each backup replica of current component
        send placement decision
```

Figure 4.5: Placement algorithm.

where the data sources are, e.g., where a packet capturing device is located in the network in the example of Figure 1.1. The source nodes are discovered by querying the DHT.

Each source node receives the user request and begins the component replica placement by deciding the placement of the primary replicas of its downstream components in the application component graph. For example, in Figure 4.4, where the application has only one source node and this node has only one downstream component, the source node decides the placement of replica c_{11}.

Phase 2. Propagation. The node that becomes the host of a primary replica is responsible for continuing the placement protocol. Becoming the host of a primary replica and consequently agreeing to continue the placement protocol is achieved by accepting a `placement request`. A placement request is sent from a node that makes a placement decision for the primary replica of the next component in the application component graph to a node that is requested to host this primary replica. A placement request includes the query plan ξ, the application component graph λ to the extent that it has been defined so far, the replication component graph ρ to the extent that it has been defined so far, the replication degree ϱ, and an index to identify which operator of the query plan the placement request refers to. In addition to hosting the requested primary component replica, the node receiving a placement request is also requested to find nodes to host this component's backup replicas and nodes to host the primary replicas of the downstream components of this component. For example, in Figure 4.4, a placement request sent from the source node to the node that is requested to host c_{11} makes the recipient responsible for finding nodes to host c_{21}, c_{31}, and c_{12}. We now describe the six detailed steps of the placement protocol that are executed on each hop for placing the primary replica of each downstream component and the backup replicas of the current component.

Step 1. Primary placement selection. A node decides upon the placement of the primary replica of each of its downstream components based on three criteria: First, nodes that have already been used for placing previous replicas for this particular application are preferred. These nodes are identified by the (partial so far) application and replication component graphs that are included in the placement request. Second, out of these previously used nodes, we select the ones that have enough residual network bandwidth to accommodate the bandwidth required by the output stream, i.e., the nodes for which $b_{s_j} \leq rb_{e_j}$. The bandwidth measurements are collected by the monitoring module, as was described in Section 4.2. The bandwidth requirement of the output stream is calculated according to the user-requested stream rate, and is included in the query plan ξ of the placement request, as was described in Section 4.2. Third, the previously used nodes that can sustain the required bandwidth are ordered from the closest to the most remote in terms of communication latency l_{e_j}. The latency measurements are again collected by the monitoring module as was described in Section 4.2. We call these nodes the *closest used candidates*. The reason we try to reuse nodes is that, as we discussed in Section 4.3.1, collocating component replicas on nodes, as much as the nodes' resources permit it, maximizes application availability. If there are not enough closest used candidates to place all the required primary replicas, closest candidates are used instead. To identify the *closest candidates*, only the last two of the above three criteria are taken into account, i.e., the residual network bandwidth and the communication latency.

Step 2. Primary placement negotiation. The placement of a primary replica of a downstream component is decided directly by a node if this node hosts its only upstream component. In Figure 4.4 for example this is the case for the host of c_{11}, which can decide the placement of c_{21} directly. However, when the downstream component has more than one upstream components, its placement decision has to be cooperative, taking into account the placement preferences of all the upstream components. The decision is made by the upper node in the application component graph, taking into account all involved nodes' placement preferences. For example, in Figure 4.4, the node hosting c_{41} can be decided by both the nodes hosting c_{21} and c_{31}. The upper node in the application component graph is defined as the decision maker, which in this case is the node hosting c_{21}. Thus, the host of c_{31} informs the host of c_{21} of its placement preferences, before the host of c_{21} can decide the placement of c_{41}.

The nodes' placement preferences are transferred using `placement negotiation` messages. A placement negotiation is sent from a node that determines that a primary replica of the next component in the application component graph can be decided by more nodes than itself, to the upper node in the graph that can make such a decision. The placement negotiation message includes ξ, λ, ρ, ϱ, an index to identify which operator of the query plan the placement negotiation refers to, and a list of nodes that the sender would want the component to be placed on, ordered by their latency to the sender. This is the list of closest used candidates the node constructs, or the list of closest candidates, if no used candidates exist. Once the recipient receives placement negotiations from all upstream components of a component that needs to be placed, it decides which node should be asked to host the downstream component. It does so by finding the first intersection of the candidate lists. The candidate lists are traversed from the list of the node with the highest requested output bandwidth to that of the node with the lowest requested output bandwidth. This way, the preferences are weighed according to the requested output bandwidth. Once the candidate for hosting the primary replica of the downstream component has been identified, either directly or through the negotiation process, a placement request is sent to it.

Step 3. Primary placement evaluation. A node receiving a placement request for hosting a primary replica evaluates whether it can accept it or not. To determine whether to accept or deny a request a node checks whether: i) The profiled processing time required for the operator of the primary replica to be instantiated will not exceed the residual processing capacity of the node, i.e., $p_{o_i} \leq rp_{v_i}$, and ii) The requested bandwidth for the output of this primary replica will not exceed the residual network bandwidth on the virtual links to the nodes that will be asked to host the downstream components of this primary replica, and to the nodes that will be asked to host its backup replicas, i.e., $b_{s_j} \leq rb_{e_j}$ for all corresponding virtual links e_j. For example, in Figure 4.4, a placement request sent from the node v_{11} hosting c_{11} to the node v_{21} that is requested to host c_{21} will be accepted only if $p_{o_2} \leq rp_{v_{21}}$, $b_{s_4} \leq rb_{e_{21\_41}}$, and $b_{s_4} \leq rb_{e_{21\_22}}$.

Both the bandwidth and the processing time requirements of a new placement are included in the query plan ξ of the placement request. Bandwidth requirements b_{s_j} are calculated according to the user-requested stream rate, while processing time requirements p_{o_i} are calculated according to the data rate and resource profiling results for the operators (Section 4.2). The residual processing capacity rp_{v_i} and the residual network bandwidth rb_{e_j} are collected by the monitoring module of the node (Section 4.2).

Once a placement request has been evaluated, the node sends a `placement reply` to the node that sent the placement request. The placement reply includes an identifier of the request it is replying to and whether the request is accepted or denied.

Step 4. Primary placement decision. The node making the placement decision of a primary replica waits for the closest used candidate's placement reply. If the placement request is denied, the next closest used candidate is contacted. If no closest used candidates accept the placement, the closest candidates are contacted.

Once a placement request is accepted, a `placement decision` is sent to the node that accepted, to complete the placement of the primary component replica. A placement decision is sent from a node that makes a placement decision for a component replica to the node that is requested to host this replica. The placement decision includes the identifier of the application the component replica will be a part of, a unique identifier of the component replica within the application, the operator the component replica will be offering, and the fact that the component will be a primary replica. The receiver of a placement decision allocates resources for the replica. This way, overallocations caused by concurrent protocol executions are avoided.

Step 5. Backup placement selection. Once a node has placed all the primary replicas of its downstream components in the application component graph, the backup replicas of the current component are placed. For example, in Figure 4.4, the host of c_{11} needs to place c_{12}, after it has placed c_{21}, and c_{31}. The backup replicas are again placed at the closest used candidates, to increase component collocation and hence maximize application availability. If the replication degree exceeds the number of closest used candidates, closest candidates are used instead. (Please note that replicas of the same component can never be collocated.) The closest used candidates are identified following the same procedure described for the placement selection of the primary replicas in step 1. The node deciding where to place a component's backup replica is hosting the primary replica that will generate the input of this backup replica. Therefore it can ensure that the requested bandwidth for the input of the backup replica can be accommodated by the residual network bandwidth on the virtual link to the node that will be asked to host it, i.e., $b_{s_j} \leq rb_{e_j}$. Again, b_{s_j} is included in the query plan ξ (Section 4.2), while rb_{e_j} is provided by the monitoring module of the node (Section 4.2). A backup replica does not have any additional requirements from the recipient regarding either processing or downstream communication. Therefore, no placement request and reply procedure similar to the primary replicas' placement is required.

Step 6. Backup placement decision. The placement of each backup component replica is completed by sending a placement decision to the node that has been decided to be the host of the backup replica. In addition to the information included in a placement decision of a primary replica, a placement decision now includes the fact that the component being placed will be a backup replica. Again, the receiver of a placement decision allocates resources for the backup replica to avoid overallocations.

Phase 3. Completion. We call the node(s) that host the outputs of the application the *destination nodes*. These nodes are usually pinned where the data receivers are, e.g., where a network operation center is located in the example of Figure 1.1. Once a node that accepted a placement request notices that it only has the destination nodes as its downstream nodes, it only has to place the backup replicas of the current component and then the placement reaches completion. The node discovers the destination nodes by querying the DHT. It then propagates the placement request for the application to the destination nodes. The placement request now includes the complete application and replication component graphs, specifying the placement of all primary and backup replicas.

The destination nodes propagate these to node v_s, which now can inform the source nodes to begin streaming.

If at any step of the placement protocol a node cannot find any candidate to host the requested component replicas, regardless of collocation (availability) and latency (performance) requirements, a failure message is returned to v_s and then to the user and the component replicas that had been placed so far are deallocated. This however is an extreme case, indicating that the system does not have the required processing and network resources to host the requested application.

Failure Handling. Failures of nodes during the protocol execution result to message timeouts, causing the sender of the corresponding message to try the next available candidate for placement. To avoid message timeouts by detecting node failures in advance, more elaborate failure detectors [81] can also be used. Moreover, failures affecting component discovery are handled by the DHT [74]. Handling failures not during the placement but during the stream processing application execution is a different and rather complicated problem, considering the request rates and the real-time requirements of the applications. Several mechanisms have been proposed to address this problem, including checkpointing [18, 39], masking [79], logging [38], and trading-off consistency [11]. Since our work focuses on placement to minimize failure probability and not on handling failures during execution, existing solutions for the latter can be integrated in our architecture.

4.5 Experimental Evaluation

We have implemented in Synergy our replica placement protocol, as well as the network monitoring application from the Stream Query Repository shown in Figure 1.1[1], and have conducted a performance evaluation over PlanetLab. To evaluate *Synergy*'s distributed placement protocol's performance, we compared it to four more placement protocols. *Optimal* places all primary component replicas on a node, and n backup replicas on each of another $\varrho - 1$ nodes. As we described in Section 4.3.2, such a placement is practically infeasible due to the processing and bandwidth constraints. However, we include it for comparison purposes, as it maximizes availability. *Random* places component replicas on nodes randomly, as is done for example in [3]. *Partition* implements a scheme [98] that aims to maximize inter-component correlation. Similar to RAID-1, it partitions nodes to groups of ϱ nodes each and assigns all replicas of a component to one group every time. [98] showed that this placement performs best for strict composite applications, therefore we include it here as the current state of the art. Finally, *Latency* places components based solely on network latencies, similarly to current placement protocols for distributed stream processing systems [4, 61] that seek to maximize application performance.

Each node in the system generates an application request that triggers component replica placement. By sharing the system resources among multiple concurrent applications, we do not give Synergy's resource-aware placement protocol an advantage over the other protocols, in terms of placement choices. In fact, due to resource sharing, Synergy results to placing the component replicas of an application to n or more nodes. We present the average

[1]Application screenshots available at http://synergy.cs.ucr.edu/screenshots.html

results over the total number of participating nodes. We look at application availability and replica failure ratio as metrics for availability, and at average inter- and intra-operator delays as metrics for performance. For clarity purposes we do not include Latency when evaluating availability, as it is equivalent to Random. Similarly, we do not include Optimal when evaluating performance, as its inter-operator delay is 0, while its intra-operator delay is equivalent to Random.

We experiment with different network sizes, percentages of failed nodes, application component graph sizes, and replication degrees, to determine the sensitivity of our results to all of these parameters. When kept constant, the values of the above parameters are 20 nodes, 3 of them failing, 9 components in the application component graph (as in Figure 1.1), and 2 replicas of each component. We artificially control node failures, while ensuring that none of the PlanetLab machines actually failed during our experiments. We chose a default failure percentage of 15% of the nodes, based on our analysis of actual ping traces between all pairs of PlanetLab nodes, obtained from [62]. By parsing these traces and considering that a node has failed whenever it is not reachable by any other node, we found 15% to be a representative failure percentage. We experiment with fail-stop failures; once a node has failed we regard all the components hosted by it, and all the applications using these components, as permanently failed.

We chose a default application component graph size of 9, capturing the graph size of the implementation of the network monitoring application from the Stream Query Repository we showed in Figure 1.1.

4.5.1 Application Availability

We first present the experimental results for the application availability achieved by the different placement protocols. We measure availability by the percentage of successful requests for composite applications.

Effect of component replication degree. When increasing the replication degree of components, application availability increases. However, an intelligent replica placement can achieve higher availability with a lower replication degree. This is shown in Figure 4.6. Synergy achieves availability close to optimal even with a replication degree of 2, by paying attention to the relative placement of replicas. In contrast, Random and Partition require one more replica to achieve comparable availability.

Effect of application component graph size. To determine the effect of the component graph size on application availability, we experimented with artificial graphs of various sizes, as shown in Figure 4.7. In general, as the size n of the application component graph increases, application availability decreases. This is because more nodes need to be employed for hosting all the component replicas (since no replicas of the same component can be hosted by the same node), while all of the components need to be available for the application to be available. However, as Figure 4.7 shows, Synergy's replica placement protocol manages to maintain high availability even in larger application component graphs. The reason lies in that Synergy reuses nodes to collocate component replicas. Hence, the number of nodes used for placement does not linearly increase as the number of components increases.

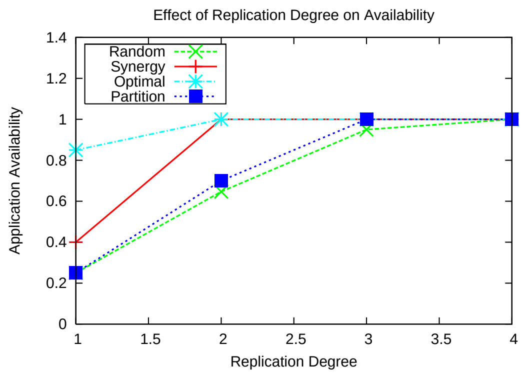


Figure 4.6: Replication degree sensitivity.

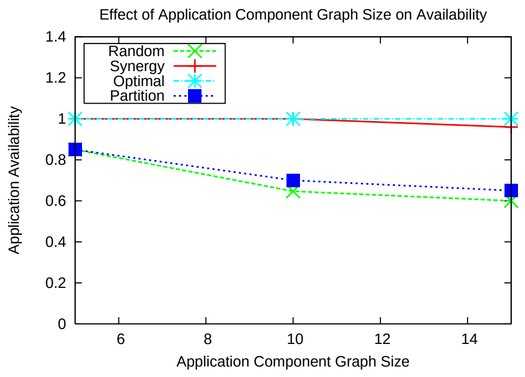


Figure 4.7: Component graph size sensitivity.

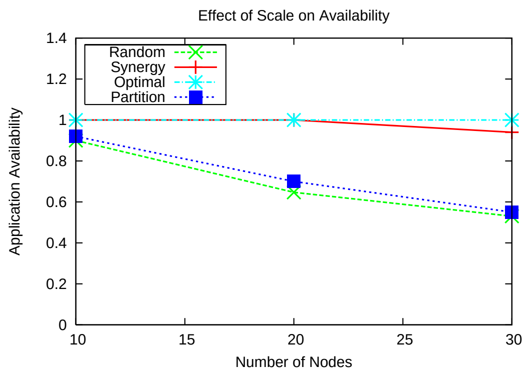


Figure 4.8: Scalability.

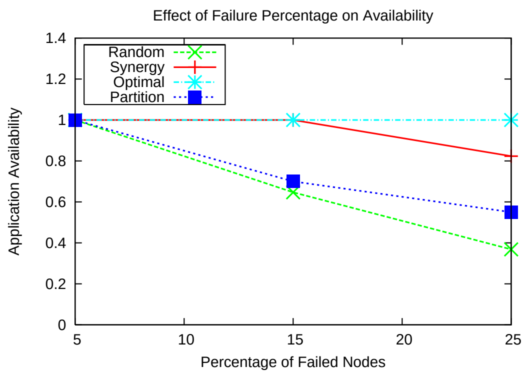


Figure 4.9: Failure percentage sensitivity.

Effect of scale. Figure 4.8 shows that the availability benefits of Synergy hold regardless of the size of the network. This is because Synergy collocates replicas on nodes that have already been used for placement as much as possible. Therefore it is not affected by the available placement options that more nodes present. This is in contrast to Random and Partition, which are blind to which nodes have already been used for placement for a particular application and have a higher probability of spreading components as more nodes are available.

Effect of failure percentage. When the percentage of failed nodes increases, inevitably availability drops. However, as Figure 4.9 shows, Synergy manages to postpone this phenomenon as much as possible, by using the minimum feasible number of nodes, thus minimizing the probability that any of the component hosts will fail. Using the minimum feasible number of nodes can only be achieved when placing components specifically catering to an application request. This is why Partition does not achieve comparable availability, since it statically places components to nodes, regardless of any particular application requests.

4.5.2 Component Replica Failure Ratio

An intelligent replica placement achieves high application availability even when the ratio of failed replicas to the number of total replicas, which we call replica failure ratio, is high. This is because just one replica of each component needs to be available. For example, Optimal can achieve availability 1 even if all nodes but one have failed, in which case the replica failure ratio is maximum ($\frac{(\varrho-1)\cdot n}{\varrho\cdot n}$). To explore this we measure the replica failure ratio and compare it to the application availability achieved by the different placement protocols. We present results from varying both the network size and the percentage of failed nodes.

Effect of scale. Figure 4.10 shows that the higher availability Synergy achieves over its competitors does in fact stem from a smaller number of replica failures. This is because Synergy's replica placement protocol tries to minimize the number of nodes

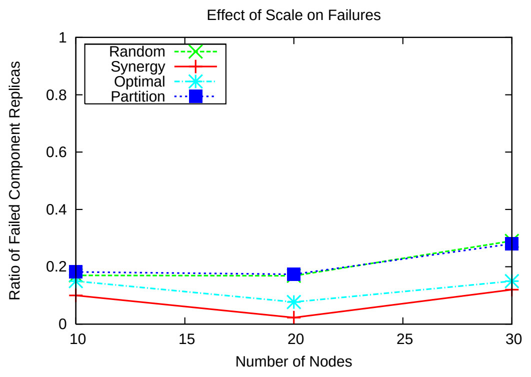


Figure 4.10: Failure ratio with scale.

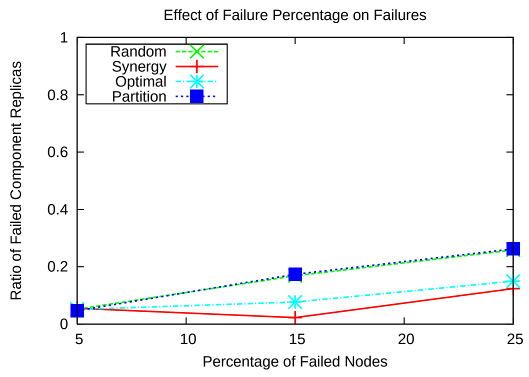


Figure 4.11: Failure ratio and percentage.

used. Optimal's placement protocol is even more intelligent, since it can achieve availability equivalent to or higher than Synergy, even though its replica failure ratio is higher.

Effect of failure percentage. The conclusions that can be drown from the effect of the percentage of failed nodes to the replica failure ratio, as shown in Figure 4.11, are similar to the ones of the effect of network scale from Figure 4.10. Moreover, we see that Partition is more appropriate as a placement strategy for distributed stream processing applications than Random, since the availability it achieves is higher, even though their replica failure ratios are similar.

4.5.3 Average Delay

We now discuss average delays attained, as they represent a measure of the performance of an instantiated application.

Effect of scale on inter-operator delay. Figure 4.12 summarizes the performance attained by a distributed stream processing application, as it is determined by the communication delay between primary component replicas. Synergy's placement protocol

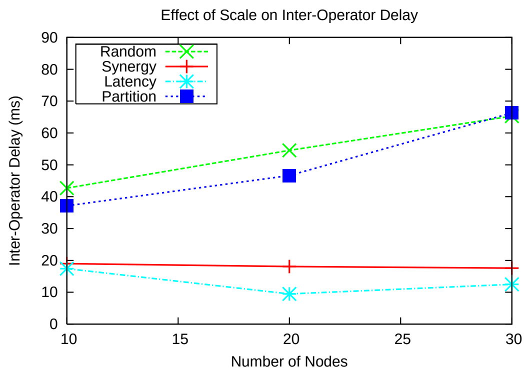


Figure 4.12: Scalability of inter-operator delay.

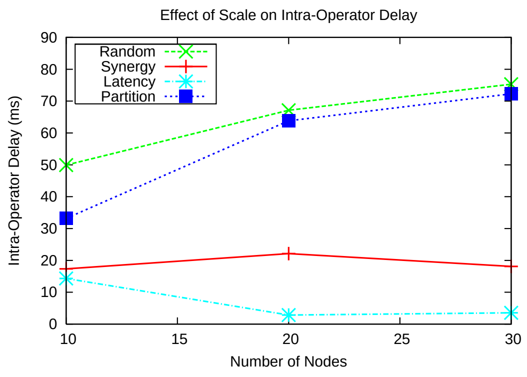


Figure 4.13: Scalability of intra-operator delay.

focuses on maximizing the availability of an application and only takes performance into account when comparing placement decisions that are equivalent in terms of availability. Yet, as Figure 4.12 shows, the performance of the applications placed with Synergy's protocol is comparable to those placed by Latency, which uses only performance as a placement criterion. As expected, Random and Partition perform much worse, since they do not consider communication delays in their placement decisions.

Effect of scale on intra-operator delay. The cost of keeping the backup replicas up to date with their primary counterparts is summarized in Figure 4.13. Again, Synergy manages to reduce the latency of these data transfers, while not sacrificing availability. Since Latency can choose the closest nodes for placement among all nodes, regardless of which have been used for placement so far, it can decrease intra-operator delay further. However, as we already discussed this leads to low availability.

Effect of scale on gathering latency information. Table 4.14 lists the average absolute time a node needs to gather latency information for virtual links to remote nodes in the overlay. This affects how fast Synergy's placement protocol can reach a decision.

Number of Nodes	Latency Information Gathering Time (ms)
10	1827
20	2100
30	5539

Figure 4.14: Latency information gathering.

As we see, the required time remains in the order of a few seconds. The fact that we try to minimize the number of nodes to use for replica placement to maximize availability also limits the number of latency measurements we need to gather.

4.6 Conclusions

In this chapter we have studied the problem of component replica placement to achieve high availability in distributed stream processing applications. This is the first work we are aware of to discuss this problem. We have identified design principles for replica placement that take into account the particular characteristics of these applications. We have incorporated these principles in a distributed replica placement protocol, that aims to maximize availability, while respecting resource constraints, and making performance-aware placement decisions. Our protocol is decentralized, allowing nodes to proceed concurrently with their placement decisions, and requiring only local knowledge. We have integrated our replica placement protocol in Synergy, our distributed stream processing middleware. Our experimental comparison over PlanetLab with the current state of the art corroborated our claims that our techniques maximize availability, while sustaining good performance.

Chapter 5

Related Work

In this chapter we discuss research efforts related to the problems we have addressed in this work. We extend our discussion in three directions, relevant to the techniques we have discussed in each chapter of the dissertation. Section 5.1 presents research works relevant to the problem of component composition (discussed in chapter 2). Section 5.2 presents work relevant to the problem of load balancing (discussed in chapter 3). Finally, Section 5.3 presents efforts related to the problem of high availability (discussed in chapter 4).

5.1 Component Composition

Distributed stream processing has been the focus of several recent research efforts from many different perspectives. In [4, 61, 78] 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 [61] mentions component reuse, they do not focus on the impact on already running applications. [46] 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 [29], while load balancing based on the correlation of the load distributions across nodes is proposed in [96]. 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 [32, 35]. 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 [20, 44, 65, 66] as well.

Application task assignment has also been the focus of many grid research efforts. GATES [21] 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 focuses on optimally assigning individual tasks to different hosts, rather than instantiating *composite* network applications. Work on resource discovery such as SWORD [58] can assist in component composition, and is thus complementary to ours.

Component composition has also been studied in the context of web services from several aspects, such as coordinating among different services to develop production workflows [84], or providing reliability through replication [14]. Similar problems are also encountered when providing dynamic web content at large scales [7], or personalized web content [23], the changing and on-demand nature of which render them more challenging than static content delivery [41].

While we focus on component composition for stream processing, our techniques may be applicable to other domains of composite applications with QoS requirements, such as QoS-sensitive web services. Similar to Synergy, works on web service composition [50, 99, 101] take into account QoS metrics. They discuss dynamic composition algorithms that select web services so that utility is maximized [50], end-to-end QoS is guaranteed [99], or maximized [101]. [99] proposes heuristics with near-optimal solutions in polynomial time, while [101] 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 [6, 35, 46, 61, 94]. However, our maximum sharing discovery and QoS impact projection algorithms are independent of the composition protocol. Therefore they could be combined with existing web service composition approaches [50, 99, 101].

5.2 Load Balancing

In distributed stream processing systems, work on the placement of components to make efficient use of resources and to maximize application performance [4, 61, 78] is complementary to ours. Any technique for deploying new components can be used, once all the nodes hosting a particular component type are overloaded. Additionally, the migration techniques presented in [61] can be used as an alternative to our migration protocol, complementing the prediction mechanisms for QoS violations presented here. Similarly, work on component composition [25, 35, 66, 68] or application adaptation [10, 20, 40, 46, 65] can assist in load balancing, complementing our migration-based solution. Load balancing for distributed stream processing applications has also been studied [12, 29, 80, 95, 96, 104]. We differ from these approaches in that we focus on the application QoS, rather than the system utilization. Furthermore, we propose a hot-spot prediction framework to drive proactive migration decisions. Load shedding [13, 85, 86, 88, 90] has been explored before as a means to alleviate application hot-spots in stream processing systems. Our goal when alleviating application hot-spots via migration is to do so in a less intrusive manner. Similar to our work, [90] identifies the need for proactive QoS management and proposes operator selectivity estimation using sampling. Their methods however refer to centralized stream processing on a single node.

Workload prediction has been studied in various contexts and [77] discusses how some workloads have been shown to be most accurately represented by open models, while others by closed ones. Dinda [24] has shown the effectiveness of linear models in predicting host load, network bandwidth, and performance data. In the domain of grid computing multi-resource prediction has been proposed [48], where the processor utilization is cross-correlated with the memory utilization. We also utilize cross-correlation, but between different nodes rather than between different resources. Performance prediction for multi-tier web servers [82, 102] is also relevant to our work, provided that all tiers are considered and not just one which is assumed to be the bottleneck. [82] proposes a model based on queuing theory, to predict performance as a function of the transaction mix. For stream processing applications however, rate fluctuations rather than the type of required processing affect performance. For the same reason, certain assumptions regarding the distribution of arrival rates that are needed for queueing analysis, may not hold. [102] proposes a model based on regression to predict the processing cost of web transactions and drive capacity planning decisions. We also employ linear regression but focus on online execution time prediction.

Load balancing in peer-to-peer systems has also been a topic of recent research [30, 103]. The focus there is on distributing objects evenly among peers, to improve resource utilization. In contrast, we focus on managing the load incurred by applications executing on top of a peer-to-peer network, in order to improve their performance.

Finally, process migration has been employed to provide dynamic load distribution, fault resilience, improved system administration, and data access locality in a variety of domains, ranging from operating systems to batch application execution and mobile agents. An overview of related research can be found in [52]. Here we explore how migration can be used to alleviate imbalances caused by fluctuations in the rates of long-running distributed applications.

5.3 High Availability

Existing research in the area of high availability for distributed stream processing systems [11, 18, 38, 39, 79] has focused on efficient replica state maintenance to mask component failures. To this extent, recovery mechanisms [38], failure masking [79], consistency trade-offs [11], and checkpoint scheduling [18,39] have been explored. In this work we focus on replica placement to maximize application availability. Therefore, techniques like the above are complementary to ours and can be integrated in our system.

Placement of components or operators has been investigated to maximize the performance of distributed stream processing systems [4,61,78]. In order to limit the number of nodes to be examined for placement, previous approaches employ heuristics that consider only a subset of all nodes [4], or employ a latency space [61]. In our case, the number of nodes to be examined for placement is limited by the fact that we want to collocate components as much as possible to maximize availability. As was already discussed in Section 4.3.1, a performance-oriented placement results to random relative replica placement with low availability.

Replica placement has been studied extensively in distributed systems, both with availability and with performance in mind. However, the focus of research in distributed storage [3, 5, 42, 75], distributed databases [28, 60, 63, 91], distributed object systems [26, 27, 31, 45, 57, 64, 93], and web services [14, 84] is on the availability of individual objects.

Similar to distributed stream processing systems, applications built on object-, component-, or service-based architectures, such as CORBA [26, 27, 31, 57] or Enterprise JavaBeans [93], or on multi-tier architectures [28, 60] are composite. While research in fault tolerance for such applications addresses timeliness and correctness in the presence of failures, it does not focus on the relative placement of objects. This is because usually an application server can host all the primary object replicas of such an application (similarly to our Optimal placement algorithm). Due to the high processing volume and rate required by distributed stream processing applications, as well as the amount of data that would have to be transferred to an individual host, this approach is usually not feasible in a distributed stream processing system. Our placement mechanisms however can be applied to distributed object systems, if the primary replicas of the objects of a composite application are distributed.

Similar to distributed stream processing applications, the applications considered in [45] have both fault tolerance and timeliness requirements. To address these needs, a two-tier replication architecture is constructed, depending on the consistency requirements of the replicas. Replica selection algorithms are then proposed to satisfy the applications' timing requirements. This way, clients that can tolerate weaker consistency can take advantage of faster service time. Unlike distributed stream processing applications however, the applications described in [45] follow a single-object paradigm, where a client request involves one object, instead of multiple.

The only study of the availability of multi-object operations in distributed systems we are aware of is [98] (with the theoretical analysis provided in [97]), which compares the availability achieved by several DHTs with regards to the strictness of an application. We are able to achieve higher availability than the protocol that is identified as best for strict operations in [98], by performing an ad-hoc placement of the replicas, once an application request arrives. Distributed stream processing applications further differ from static distributed applications, in that replicas communicate with each other. This includes communication both between primaries as well as between primaries and backups. This communication affects application performance and therefore is taken into account by our placement protocol.

Chapter 6

Conclusions and Future Work

In this dissertation we have discussed techniques for providing Quality of Service (QoS) support for distributed stream processing applications. We have also presented the implementation of the techniques we propose in Synergy, our peer-to-peer distributed stream processing middleware. We have extended our efforts towards three directions. First, we have discussed how end-to-end delay QoS requirements can be taken into account when composing a new application. Second, we have discussed how an application can continue adhering to these requirements while executing. Third, we looked at availability as a QoS metric and we have discussed placement of component replicas to maximize it.

6.1 Concluding Remarks

Synergy's sharing-aware component composition algorithms are completely decentralized and focus on reusing both streams and components. We have proposed QoS projection to ensure that the QoS requirements of the currently running applications will not be violated when admitting a new application. Synergy's load management architecture focuses on application hot-spot prediction and alleviation. The algorithms we propose for hot-spot prediction are based on the statistical methods of linear regression and correlation, utilizing only light-weight, passive measurements. Statistics collection and hot-spot prediction and alleviation are carried out at run-time by all nodes independently, building upon a fully decentralized architecture. To alleviate hot-spots, Synergy empowers nodes to autonomously migrate the execution of stream processing components using a migration protocol that offers minimal disruption in the application execution. Finally, Synergy employs a distributed replica placement protocol that aims to maximize application availability, while respecting resource constraints, and making performance-aware placement decisions. The protocol is decentralized, allowing nodes to proceed concurrently with their placement decisions, and requiring only local knowledge.

We have implemented our techniques in a software prototype of Synergy, a multi-threaded system of about 35,000 lines of Java code. Deploying the system over PlanetLab using a real network monitoring application operating on traces of real streaming data has enabled us to conduct a thorough experimental evaluation of our techniques. More information on the Synergy middleware can be found at http://synergy.cs.ucr.edu

6.2 Future Directions

Our future work includes the integration of iterative execution of Synergy's composition protocol with techniques for increasing application reliability. This can enable distributed stream processing systems 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. To offer fault tolerance, either reactive or proactive failure recovery schemes can be used [33]. 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.

Adhering to QoS requirements by eliminating hot-spots is an important step towards dependable distributed stream processing applications. In this work we have focused on shared processing power as a cause for application hot-spots. As part of our future work we plan to investigate extending our architecture to include multiple shared system resources, such as bandwidth, memory, or storage, in addition to the processor. State transfer when migrating complex components is another important area of future work.

It would also be interesting to incorporate to our middleware current research on fault tolerant distributed stream processing systems, such as checkpointing techniques for consistency maintenance and failover techniques for failure masking. Finally Synergy's replica placement protocol could be integrated with performance-oriented placement protocols, which would include maximizing the availability of already deployed application component graphs.

We have also identified several directions towards long-term future work. A key challenge for the future will be to provide distributed systems that can keep up with the current user growth, while providing QoS guarantees. This translates to designing systems that can cope with the data volumes generated by Internet-scale applications, while providing the real-time and highly available services users are accustomed to.

Replication of stream processing components can alleviate performance bottlenecks or also increase the fault-tolerance of distributed stream processing applications. However, when splitting the processing load among multiple components, a fundamental trade-off exists: Consistent replication, in which components have an accurate view of each other's state at all times incurs high synchronization and communication overheads. This is particularly true for stream processing systems, that deal with high volumes of data that are updated continuously. Existing stream sketching and aggregation techniques can possibly be applied on replication and integrated with consistency protocols. Such techniques can enable accurate state reconstruction, while minimizing the amount of data that needs to be transferred between replicas. Furthermore, existing consistency protocols may need to be revisited to apply on such highly loaded environments, and it would be interesting to investigate what requirements can be relaxed with minimal effect on application accuracy, to enable efficient replication. Since a stream processing application is described by a graph, graph theory can help determining which components will benefit most the overall performance if replicated.

As human supervision has evolved into the major factor of the total cost of IT operations, designing large-scale systems that can achieve QoS goals while being easy to manage will be of key importance. Enabling systems to make adaptive decisions regarding load and topology management can be an important step in that direction. While load management has been the focus of several research efforts, several issues remain open with regards to topology management. When overlay networks are employed, the discrepancy between overlay and physical routing, also known as overlay stretch, can affect considerably the application performance. Even when direct connections between nodes are employed, they need to be decided adaptively, according to the currently executing applications. Making these decisions at a large scale is non-trivial. For example, in a distributed stream processing system, each node hosts multiple components, each of which participates in several applications. Ideally, the connections between nodes should match the component interactions, as they evolve. Especially for stream processing applications, in which large volumes of data are transferred between nodes, topology can affect application performance in a crucial way. By making use of data mining and machine learning techniques to model component interactions, it would be interesting to identify communication patterns, and drive adaptive topology management decisions.

Incorporating security, trust, and privacy requirements in the design of distributed stream processing systems is another important parameter to take into account, as multiple entities representing businesses or individual users engage in high volumes of unsupervised transactions. Sharing data and using remote services while abiding to access control requirements poses significant challenges in loosely coupled, unstructured, and large-scale distributed systems. It would be interesting to investigate how applications invoking multiple services hosted by different nodes can be composed while respecting data access restrictions. To achieve this cryptographic techniques such as private information retrieval and zero-knowledge proofs may be adapted to specific application domains, and virtualization and Service-Oriented Architecture can help in enforcing isolation and autonomy requirements.

Bibliography

[1] D.J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.H. Hwang, W. Lindner, A.S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. In *Proceedings of the Second Biennial Conference on Innovative Data Systems Research, CIDR, Asilomar, CA, USA*, January 2005.

[2] T.F. Abdelzaher. An automated profiling subsystem for QoS-aware services. In *Proceedings of the 6th IEEE Real-Time Technology and Applications Symposium, RTAS, Washington, DC, USA*, June 2000.

[3] A. Adya et al. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation, OSDI, Boston*, December 2002.

[4] Y. Ahmad and U. Çetintemel. Network-aware query processing for stream-based applications. In *Proceedings of the 30th International Conference on Very Large Data Bases, VLDB, Toronto, Canada*, August 2004.

[5] A.S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.P. Martin, and C. Porth. BAR fault tolerance for cooperative services. In *Proceedings of 20th Symposium on Operating Systems Principles, SOSP, Brighton, UK*, October 2005.

[6] L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure. Adaptive control of extreme-scale stream processing systems. In *Proceedings of the 26th International Conference on Distributed Computing Systems, ICDCS, Lisboa, Portugal*, July 2006.

[7] C. Amza, A.L. Cox, and W. Zwaenepoel. A comparative evaluation of transparent scaling techniques for dynamic content servers. In *Proceedings of the 21st International Conference on Data Engineering, ICDE, Tokyo, Japan*, April 2005.

[8] N. Antunes, C. Fricker, F. Guillemin, and Ph. Robert. Integration of streaming services and TCP data transmission in the Internet. *Elsevier Performance Evaluation*, 62(1–4):263–277, October 2005.

[9] M.J. Karam A.P. Markopoulou, F.A. Tobagi. Assessing the quality of voice communications over Internet backbones. *IEEE/ACM Transactions on Networking*, 11(5):747–760, October 2003.

[10] R.H. Arpaci-Dusseau. Run-time adaptation in river. *ACM Transactions on Computer Systems*, 21(1):36–86, February 2003.

[11] 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.

[12] M. Balazinska, H. Balakrishnan, and M. Stonebraker. Contract-based load management in federated distributed systems. In *Proceedings of Networked System Design and Implementation, NSDI, San Francisco, CA, USA*, March 2004.

[13] P. Barlet-Ros, G. Iannaccone, J. Sanjus-Cuxart, D. Amores-Lpez, and J. Sol-Pareta. Load shedding in network monitoring applications. In *Proceedings of the 2007 USENIX Annual Technical Conference, Santa Clara, CA, USA*, June 2007.

[14] A. Bartoli, R. Jimenez-Peris, B. Kemme, C. Pautasso, S. Patarin, S. Wheater, and S. Woodman. The ADAPT framework for adaptable and composable web services. *IEEE Distributed Systems On Line*, Web Systems Section, September 2005.

[15] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, L. Peterson, T. Roscoe, and M. Wawrzoniak. Operating systems support for planetary-scale network services. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation, NSDI, San Francisco, CA, USA*, March 2004.

[16] K.P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, December 1993.

[17] N. Budhlraja, K. Marzullo, F. B. Schneider, and S. Toueg. Primary-Backup protocols: Lower bounds and optimal implementations. In *Cornell University Technical Report TR-92-1265*, January 1992.

[18] Z. Cai, V. Kumar, B.F. Cooper, G. Eisenhauer, K. Schwan, and R.E. Strom. Utility-driven proactive management of availability in enterprise-scale information flows. In *Proceedings of 7th Middleware, Melbourne*, November 2006.

[19] S. Chandrasekaran, O. Cooper, A. Deshpande, M.J. Franklin andJ.M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and Mehul Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research, CIDR, Asilomar, CA, USA*, January 2003.

[20] F. Chen, T. Repantis, and V. Kalogeraki. Coordinated media streaming and transcoding in peer-to-peer systems. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium, IPDPS, Denver, CO, USA*, April 2005.

[21] L. Chen, K. Reddy, and G. Agrawal. GATES: A grid-based middleware for distributed processing of data streams. In *Proceedings of the 13th IEEE International Symposium on High-Performance Distributed Computing, HPDC-13, Honolulu, HI, USA*, June 2004.

[22] 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*, 12(5):781–794, October 2004.

[23] M. Colajanni, R. Grieco, D. Malandrino, F. Mazzoni, and V. Scarano. A scalable framework for the support of advanced edge services. In *Proceedings of the 2005 International Conference on High Performance Computing and Communications, HPCC-05, Sorrento, Italy*, September 2005.

[24] P.A. Dinda. Design, implementation, and performance of an extensible toolkit for resource prediction in distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 17(2):160–173, February 2006.

[25] Y. Drougas, T. Repantis, and V. Kalogeraki. Load balancing techniques for distributed stream processing applications in overlay environments. In *Proceedings of the 9th International Symposium on Object and Component-Oriented Real-Time Distributed Computing, ISORC, Gyeongju, Korea*, April 2006.

[26] P. Felber, B. Garbinato, and R. Guerraoui. The design of a CORBA group communication service. In *Proceedings of 15th Symposium on Reliable Distributed Systems, SRDS, Ontario, Canada*, October 1996.

[27] P. Felber and P. Narasimhan. Experiences, approaches and challenges in building fault-tolerant CORBA systems. *IEEE Transactions on Computers*, 54(5):497–511, May 2004.

[28] S. Frølund and R. Guerraoui. e-Transactions: End-to-end reliability for three-tier architectures. *IEEE Transactions on Software Engineering*, 28(4):378–395, April 2002.

[29] B. Gedik and L. Liu. PeerCQ: A decentralized and self-configuring peer-to-peer information monitoring system. In *Proceedings of the 23rd International Conference on Distributed Computing Systems, ICDCS, Providence, RI, USA*, May 2003.

[30] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in dynamic structured P2P systems. In *Proceedings of INFOCOM, Hong Kong*, March 2004.

[31] A. Gokhale, B. Natarajan, D.C. Schmidt, and J. Cross. Towards real-time fault-tolerant CORBA middleware. *Cluster Computing*, 7(4):331–346, October 2004.

[32] X. Gu and K. Nahrstedt. Distributed multimedia service composition with statistical QoS assurances. *IEEE Transactions on Multimedia*, 8(1):141–151, February 2006.

[33] X. Gu and K. Nahrstedt. On composing stream applications in peer-to-peer environments. *IEEE Transactions on Parallel and Distributed Systems*, 17(8):824–837, July 2006.

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

[35] X. Gu, P.S. Yu, and K. Nahrstedt. Optimal component composition for scalable stream processing. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems, ICDCS, Columbus, OH, USA*, June 2005.

[36] M.A. Hammad, M.J. Franklin, W.G. Aref, and A.K. Elmagarmid. Scheduling for shared window joins over data streams. In *Proceedings of the 29th International Conference on Very Large Data Bases, VLDB, Berlin, Germany*, September 2003.

[37] N. Hu and P. Steenkiste. Exploiting internet route sharing for large scale available bandwidth estimation. In *Proceedings of Internet Measurement Conference, IMC, New Orleans, LA*, October 2005.

[38] J.H. Hwang, M. Balazinska, A. Rasin, U. Çetintemel, M. Stonebraker, and S. Zdonik. High-availability algorithms for distributed stream processing. In *Proceedings of 21st International Conference on Data Engineering, ICDE, Tokyo, Japan*, April 2005.

[39] J.H. Hwang, Y. Xing, U. Çetintemel, and S. Zdonik. A cooperative, self-configuring high-availability solution for stream processing. In *Proceedings of 23rd International Conference on Data Engineering, ICDE, Istanbul, Turkey*, April 2007.

[40] V. Kalogeraki, F. Chen, T. Repantis, and D. Zeinalipour-Yazti. Towards self-managing qos-enabled peer-to-peer systems. *Self-Star Properties in Complex Information Systems, Hot Topics in Computer Science, Springer Lecture Notes in Computer Science*, 3460:325–342, May 2005.

[41] P. Karbhari, M. Rabinovich, Z. Xiao, and F. Douglis. ACDN: A content delivery network for applications. In *Proceedings of 21st ACM SIGMOD Conference, Madison, WI, USA*, June 2002.

[42] A.M. Kermarrec and C. Morin. Smooth and efficient integration of high-availability in a parallel single level store system. In *Proceedings of Euro-Par*, August 2001.

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

[44] F. Kon, R. Campbell, and K. Nahrstedt. Using dynamic configuration to manage a scalable multimedia distributed system. *Computer Communications Journal*, 24:105–123, 2001.

[45] S. Krishnamurthy, W.H. Sanders, and M. Cukier. An adaptive quality of service aware middleware for replicated services. *IEEE Transactions on Parallel and Distributed Systems*, 14(11):1112–1125, November 2003.

[46] V. Kumar, B.F. Cooper, Z. Cai, G. Eisenhauer, and K. Schwan. Resource-aware distributed stream management using dynamic overlays. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems, ICDCS, Columbus, OH, USA*, June 2005.

[47] J. Ledlie, P. Gardner, and M. Seltzer. Network coordinates in the wild. In *Proceedings of Networked System Design and Implementation (NSDI), Cambridge, MA, USA*, April 2007.

[48] J. Liang, K. Nahrstedt, and Y. Zhou. Adaptive multi-resource prediction in distributed resource sharing environment. In *Proceedings of the 4th International Symposium on Cluster Computing and the Grid, CCGRID, Chicago, IL, USA*, April 2004.

[49] P.M. Melliar-Smith and L.E. Moser. Surviving network partitioning. *IEEE Computer*, 31(3):62–68, March 1998.

[50] D. Menasce. Composing web services: A QoS view. *IEEE Internet Computing*, 8(6):88–90, November/December 2004.

[51] M.G. Merideth, A. Iyengar, T.A. Mikalsen, S. Tai, I. Rouvellou, and P. Narasimhan. Thema: Byzantine-fault-tolerant middleware for web-service applications. In *Proceedings of 24th Symposium on Reliable Distributed Systems, SRDS, Orlando, FL*, October 2005.

[52] D.S. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process migration. *ACM Computing Surveys*, 32(3):241–299, September 2000.

[53] D.C. Montgomery and G.C. Runger. *Applied Statistics and Probability for Engineers.* John Wiley & Sons Inc., New York, NY, USA, 2006.

[54] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research, CIDR, Asilomar, CA, USA*, January 2003.

[55] National Institute of Science and Technology. Secure Hash Standard (SHA1). Federal Information Processing Standard, FIPS 180-1, April 1995.

[56] NLANR/DAST Iperf. `http://dast.nlanr.net/Projects/Iperf/`, 2005.

[57] Object Management Group. Fault tolerant CORBA. *OMG Technical Committee Document formal /02-06-59, Chapter 23, CORBA/IIOP 3.0.3*, 2004.

[58] D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat. Design and implementation tradeoffs for wide-area resource discovery. In *Proceedings of 14th IEEE International Symposium on High-Performance Distributed Computing, HPDC-14, Research Triangle Park, NC, USA*, July 2005.

[59] D. Oppenheimer, B. Chun, D. Patterson, A.C. Snoeren, and A. Vahdat. Service placement in a shared wide-area platform. In *Proceedings of the 2006 USENIX Annual Technical Conference, Boston, MA, USA*, June 2006.

[60] M. Patino-Martinez, R. Jimenez-Peris, B. Kemme, and G. Alonso. Consistent database replication at the middleware level. *ACM Transactions on Computers*, 23(4):1–49, 2005.

[61] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE, Atlanta, GA, USA*, April 2006.

[62] PlanetLab All Pairs Pings. `http://pdos.csail.mit.edu/~strib/pl_app/`, 2005.

[63] C. Plattner, G. Alonso, and M. T. zsu. DBFarm: A scalable cluster for multiple databases. In *Proceedings of 7th Middleware, Melbourne, Australia*, November 2006.

[64] Y. Ren, D.E. Bakken, T. Courtney, M. Cukier, D.A. Karr, P. Rubel, C. Sabnis, W.H. Sanders, R.E. Schantz, and M. Seri. AQuA: An adaptive architecture that provides dependable distributed objects. *IEEE Transactions on Computers*, 52(1):31–50, January 2003.

[65] T. Repantis, F. Chen, and V. Kalogeraki. Cooperative media processing and streaming. In *7th Annual Industry Day Poster Session, University of California, Riverside, Second Best Graduate Poster Award*, October 2005.

[66] T. Repantis, Y. Drougas, and V. Kalogeraki. Adaptive resource management in peer-to-peer middleware. In *Proceedings of the 13th International Workshop on Parallel and Distributed Real-Time Systems, WPDRTS, Denver, CO, USA*, April 2005.

[67] T. Repantis, X. Gu, and V. Kalogeraki. QoS-aware shared component composition for distributed stream processing systems. *IEEE Transactions on Parallel and Distributed Systems, TPDS*, To appear.

[68] T. Repantis, X. Gu, and V. Kalogeraki. Synergy: Sharing-aware component composition for distributed stream processing systems. In *Proceedings of the 7th ACM/IFIP/USENIX International Middleware Conference, MIDDLEWARE, Melbourne, Australia*, November 2006.

[69] T. Repantis, X. Gu, and V. Kalogeraki. Synergy: A distributed stream processing middleware. In *Graduate Research Awards and Colloquium, University of California, Riverside, Honorable Mention*, June 2007.

[70] T. Repantis and V. Kalogeraki. Alleviating hot-spots in peer-to-peer stream processing environments. In *Proceedings of the 5th International Workshop on Databases, Information Systems and Peer-to-Peer Computing, DBISP2P, Vienna, Austria*, September 2007.

[71] T. Repantis and V. Kalogeraki. Hot-spot prediction and alleviation in distributed stream processing applications. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN, Anchorage, AL, USA*, June 2008.

[72] T. Repantis and V. Kalogeraki. Replica placement for high availability in distributed stream processing systems. In *Proceedings of the 2nd International Conference on Distributed Event-Based Systems (DEBS), Rome, Italy*, July 2008.

[73] T. Repantis, V. Kalogeraki, and X. Gu. Synergy: Quality of service support for distributed stream processing systems. In *Graduate Research Awards and Colloquium, University of California, Riverside, Graduate Research Award*, June 2008.

[74] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms, MIDDLEWARE, Heidelberg, Germany*, November 2001.

[75] F. Schintke and A. Reinefeld. Modeling replica availability in large data grids. *Grid Computing*, 1(2):219–227, June 2003.

[76] F.B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[77] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open versus closed: A cautionary tale. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation, NSDI, San Jose, CA, USA*, May 2006.

[78] S. Seshadri, V. Kumar, and B.F. Cooper. Optimizing multiple queries in distributed data stream systems. In *Proceedings of the 2nd IEEE International Workshop on Networking Meets Databases, NetDB, Atlanta, GA, USA*, April 2006.

[79] M.A. Shah, J.M. Hellerstein, and E. Brewer. Highly available, fault-tolerant, parallel dataflows. In *Proceedings of ACM SIGMOD, Paris, France*, June 2004.

[80] M.A. Shah, J.M. Hellerstein, S.Chandrasekaran, and M.J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Proceedings of the 19th International Conference on Data Engineering, ICDE, Bangalore, India*, March 2003.

[81] K.C.W. So and E.G. Sirer. Latency and bandwidth-minimizing failure detectors. In *Proceedings of 2nd EuroSys Conference, Lisboa, Portugal*, March 2007.

[82] C. Stewart, T. Kelly, and A. Zhang. Exploiting nonstationarity for performance prediction. In *Proceedings of EuroSys, Lisbon, Portugal*, March 2007.

[83] Stream Query Repository: Network Traffic Management. http://infolab.stanford.edu/stream/sqr/netmon.html, 2002.

[84] S. Tai, R. Khalaf, and T. Mikalsen. Composition of coordinated web services. In *Proceedings of the ACM/IFIP/USENIX 5th International Middleware Conference, Toronto, Canada*, October 2004.

[85] N. Tatbul, U. Cetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *Proceedings of the 29th International Conference on Very Large Data Bases, VLDB, Berlin, Germany*, September 2003.

[86] N. Tatbul, U. Cetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *Proceedings of the 29th International Conference on Very Large Data Bases, VLDB, Berlin, Germany*, pages 309–320, September 2003.

[87] The Internet Traffic Archive. http://ita.ee.lbl.gov/html/traces.html, 1994.

[88] Y.C. Tu, S. Liu, S. Prabhakar, and B. Yao. Load shedding in stream databases: A control-based approach. In *Proceedings of the 32nd International Conference on Very Large Data Bases, VLDB, Seoul, Korea*, pages 787–798, September 2006.

[89] UC Berkeley Sonoma Dust. http://www.cs.berkeley.edu/~get/sonoma/data.html, 2004.

[90] Y. Wei, V. Prasad, S.H. Son, and J.A. Stankovic. Prediction-based QoS management for real-time data streams. In *Proceedings of the 27th IEEE Real-Time Systems Symposium, RTSS, Rio de Janeiro, Brazil*, December 2006.

[91] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proceedings of 20th IEEE International Conference on Distributed Computing Systems, ICDCS, Taipei, Taiwan*, April 2000.

[92] B. Wong, A. Slivkins, and E.G. Sirer. Meridian: A lightweight network location service without virtual coordinates. In *Proceedings of ACM SIGCOMM, Philadelphia, PA, USA*, August 2005.

[93] H. Wu and B. Kemme. Fault-tolerance for stateful application servers in the presence of advanced transaction patterns. In *Proceedings of 24th Symposium on Reliable Distributed Systems, SRDS, Orlando, FL*, October 2005.

[94] K.L. Wu, P.S. Yu, B. Gedik, K.W. Hildrum, C.C. Aggarwal, E. Bouillet, W. Fan, D.A. George, X. Gu, G. Luo, and H. Wang. Challenges and experience in prototyping a multi-modal stream analytic and monitoring application on System S. In *Proceedings of the 33rd International Conference on Very Large Data Bases, VLDB, Vienna, Austria*, September 2007.

[95] Y. Xing, J.H. Hwang, U. Cetintemel, and S. Zdonik. Providing resiliency to load variations in distributed stream processing. In *Proceedings of the 32nd International Conference on Very Large Data Bases, VLDB, Seoul, Korea*, September 2006.

[96] Y. Xing, S. Zdonik, and J.H. Hwang. Dynamic load distribution in the Borealis stream processor. In *Proceedings of the 21st International Conference on Data Engineering, ICDE, Tokyo, Japan*, April 2005.

[97] H. Yu and P.B. Gibbons. Optimal inter-object correlation when replicating for availability. In *Proceedings of 26th Symposium on Principles of Distributed Computing, PODC, Portland, OR, USA*, August 2007.

[98] H. Yu, P.B. Gibbons, and S. Nath. Availability of multi-object operations. In *Proceedings of 3rd Symposium on Networked Systems Design and Implementation, NSDI, San Jose, CA, USA*, May 2006.

[99] T. Yu, Y. Zhang, and K.J. Lin. Efficient algorithms for web services selection with end-to-end QoS constraints. *ACM Transactions on the Web*, 1(1):1–26, May 2007.

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

[101] L. Zeng, B. Benatallah, A.H.H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. QoS-aware middleware for web services composition. *IEEE Transactions on Software Engineering*, 30(5):311–327, May 2004.

[102] Q. Zhang, L. Cherkasova, and E. Smirni. A regression-based analytic model for dynamic resource provisioning of multi-tier appications. In *Proceedings of the 4th IEEE International Conference on Autonomic Computing, ICAC, Jacksonville, FL, USA*, June 2007.

[103] X. Zhou and W. Nejdl. Priority based load balancing in a self-interested P2P network. In *Proceedings of the 4th International Workshop on Databases, Information Systems and P2P Computing, DBISP2P, Seoul, Korea*, pages 355–367, September 2006.

[104] Y. Zhou, B.C. Ooi, and K.L. Tan. Dynamic load management for distributed continuous query systems. In *Proceedings of the 21st International Conference on Data Engineering, ICDE, Tokyo, Japan*, April 2005.