Replica Placement for High Availability in Distributed Stream Processing Systems

Thomas Repantis Vana Kalogeraki
Department of Computer Science & Engineering
University of California, Riverside
Riverside, CA 92521
{trep,vana}@cs.ucr.edu

ABSTRACT

A significant number of emerging on-line data analysis applications require the processing of data streams, large amounts of data that get updated continuously, to generate outputs of interest or to identify meaningful events. Example domains include network traffic management, stock price monitoring, customized e-commerce websites, and analysis of sensor data. In this paper we look at the problem of high availability in such 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 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 corroborates our claims that our techniques maximize availability while sustaining good performance.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems

General Terms

Algorithms, Design, Experimentation, Reliability

Keywords

Distributed Stream Processing, High Availability, Replica Placement

1. INTRODUCTION

Over the past few years there is an increasing number of event-based systems that deal with large volume and high rate data feeds. In such systems data streams are processed in or near real-time for a variety of purposes, such as monitoring or on-line decision making. Application domains include network traffic management, financial trades surveillance, customized e-commerce applications, and analysis of sensor data.

To facilitate the performance and scalability requirements of these applications, a number of distributed stream processing systems have been proposed (e.g., [1, 3, 23, 26, 37]). These 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 [3, 23], in this paper 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 [15], failure masking [32], consistency trade-offs [5], and scheduling of checkpoints [10, 16] have been investigated.

We use a simple stream processing application example from the Stream Query Repository[1] to illustrate the availability requirements of distributed stream processing applications. In our 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 shows the components that are involved in processing the packet input over 20-minute windows, to generate the monitoring output.

Despite its simplicity this example 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. We call such applications *composite*. ii) More importantly, a distributed stream processing application such as the one of Figure 1 cannot tolerate availability of a subset of the compo-

© ACM, 2008. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in 978-1-60558-090-6/08/07, DEBS 2008. http://doi.acm.org/10.1145/1385989.1386012

[1]http://www-db.stanford.edu/stream/sqr/netmon.html

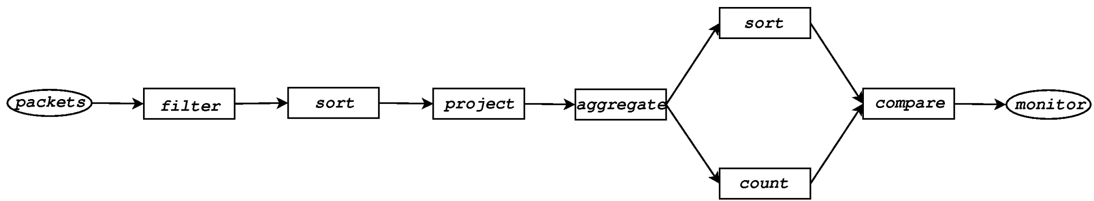

Figure 1: Stream processing application example.

nents. 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 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. 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 [13, 22, 24, 34], distributed object systems [11, 12, 14, 18, 21, 25, 36], and web services [6, 20]. While many aspects of replication have been studied extensively, and solutions such as active [31] and passive [9] 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 [2, 4, 17, 30], or databases [13, 22, 24, 34]. 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 [11, 12, 14, 21], Enterprise JavaBeans [36], or multi-tier architectures [13, 22]. 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.

This paper addresses the problem of component replica placement to maximize the availability of distributed stream processing applications, by making the following contributions:

- 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 application performance. While component placement to maximize the performance of distributed stream processing applications has been investigated before [3, 23], to the best of our knowledge this is the first paper 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.

- We incorporate our replica placement protocol for high availability in Synergy [26], our distributed stream processing middleware. We evaluate the performance of our prototype on the PlanetLab [7] wide-area network testbed. To assess the availability gains of our protocol, we compare it to the current state of the art for replica placement for high availability of multi-object operations in general distributed systems [39]. Additionally, we compare our protocol to an optimal and to a random replica placement. Finally, we compare the application performance we achieve to the performance attained by a placement protocol that focuses only on application performance, such as the existing placement protocols for distributed stream processing systems [3, 23]. Our results show that our protocol achieves availability close to optimal, outperforming its competitors, while sustaining performance close to that of a performance-oriented placement.

2. MIDDLEWARE OVERVIEW

We have implemented our replica placement protocol in the Synergy distributed stream processing middleware [26]. We begin by presenting an overview of the middleware. Figure 2 shows the architecture of Synergy, built on a peer-to-peer overlay. The middleware's goal is to support the execution of distributed stream processing applications with QoS constraints, while efficiently managing the system's resources.

As illustrated in Figure 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 registering 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) [29]. 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 and data streams between nodes. 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

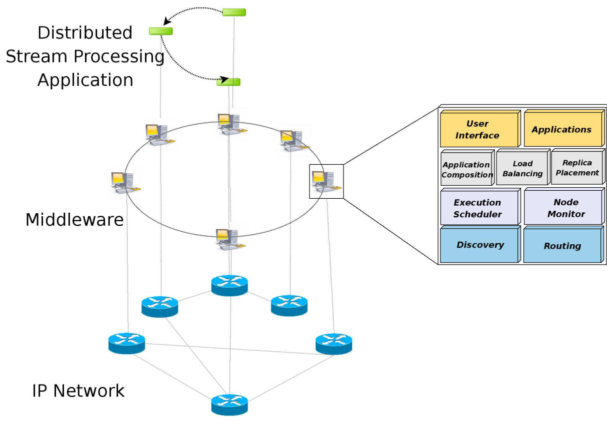

Figure 2: Middleware architecture.

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) A *composition module* that composes stream processing applications from already deployed components. The middleware adopts a fully decentralized architecture, where any node can compose a distributed stream processing application [26]. vi) A *load balancing module* that proactively alleviates hot-spots relying on decentralized migration decisions [27,28]. vii) 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. viii) A *user interface module* to control and monitor the execution of the middleware and of the applications. ix) Finally, we have extended Synergy with a *replica placement module*, that we describe in this paper, that determines on which nodes to place the component replicas of a composite application to maximize application availability.

3. SYSTEM MODEL

We now define our system model. Table 1 summarizes our notation. Each 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). 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 [35].

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*.

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 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 [26]. 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 [9,12]. 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 paper and existing solutions for consistency, checkpointing, failure masking, and recovery for distributed stream processing systems [5,15,32], as well as solutions based on view-synchronous communication [8], 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

Notation	Meaning	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

Table 1: Notations.

of the DHT substrate, or by some other mechanism that guarantees eventual consistency [5, 19]. 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.

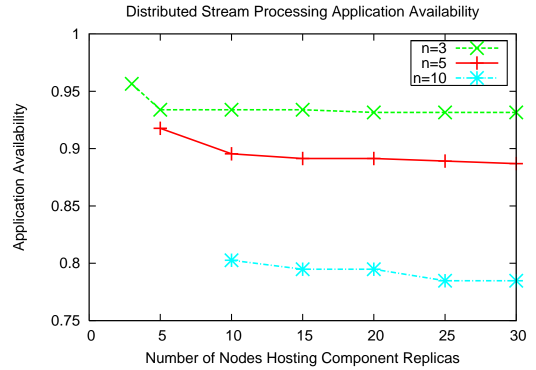

Figure 3: Availability decreases with larger application component graphs and increases as components are concentrated in fewer nodes.

4. 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 1, namely: i) Determining a placement of component replicas that maximizes availability (section 4.1). ii) Determining the number of nodes to use for placing the component replicas according to the system's resource availability (section 4.2). iii) Determining where to place the component replicas across nodes to maximize application performance (section 4.3).

4.1 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 [2,4,17,30] or client/server computing [6, 12, 14, 18, 20, 25, 36] 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 $\varrho = 2$ 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 3 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 6 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 [3, 23] 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, [39], 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, [39] 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 6 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 [39]. 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 [39] (e.g., storage).

4.2 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 3, 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 3.

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 5, 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 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 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 5 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 $c_{inter} = \sum_{j \in 1 \ldots n} s_j \cdot l_{e_j}$ 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 $c_{intra} = \sum_{i \in 1 \ldots n} \sum_{x \in 1 \ldots \varrho} s_{ix} \cdot l_{ix}$ 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}.

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 [3, 23] 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

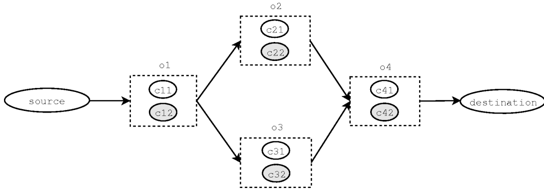

Figure 4: A simple distributed stream processing application.

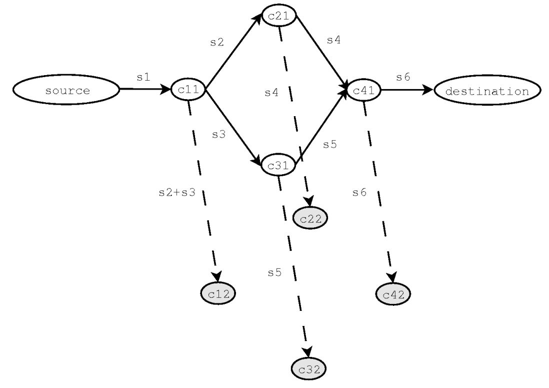

Figure 5: Replication in a simple distributed stream processing application.

solution is an NP-complete problem [3].

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.

5. 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 4.1), ii) the number of nodes to use for placement so that system resources are not exceeded (section 4.2), and iii) which nodes to use so that the communication costs are minimized (section 4.3). 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 Algorithm 1.

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 where the data sources are, e.g., where a packet capturing device is located in the network in the example of Figure 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 5, 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 5, a placement request sent from the source node to the node that is requested to host

Algorithm 1 Placement algorithm.

Input: query plan ξ, replication degree ϱ, node v_s
Output: application component graph λ,
replication component graph ρ
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

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 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 3. 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 2. We call these nodes the *closest used candidates*. The reason we try to reuse nodes is that, as we discussed in section 4.1, 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 5 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 5, 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 5, 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 3). 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 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 5, 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 3), while rb_{e_j} is provided by the monitoring module of the node (section 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. 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 [33] can also be used. Moreover, failures affecting component discovery are handled by the DHT [29]. 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 [10, 16], masking [32], logging [15], and trading-off consistency [5]. 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.

6. 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, and have conducted a performance evaluation over PlanetLab. Synergy is implemented as a multi-threaded system of about 35,000 lines of Java code. 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 4.2, 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 [2]. *Partition* implements a scheme [39] 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. [39] 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 [3,23] 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 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), 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 http://pdos.csail.mit.edu/~strib/pl_app/. 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

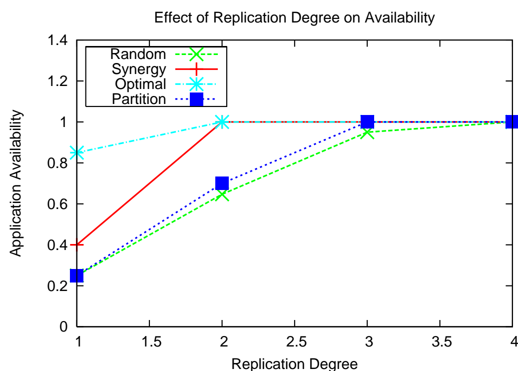

Figure 6: Replication degree sensitivity.

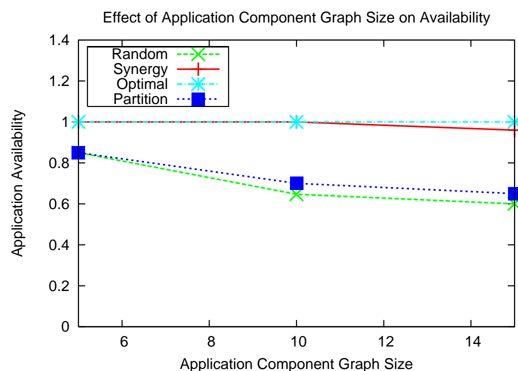

Figure 7: Component graph size sensitivity.

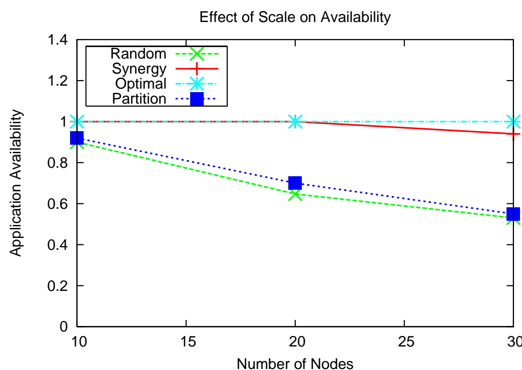

Figure 8: Scalability.

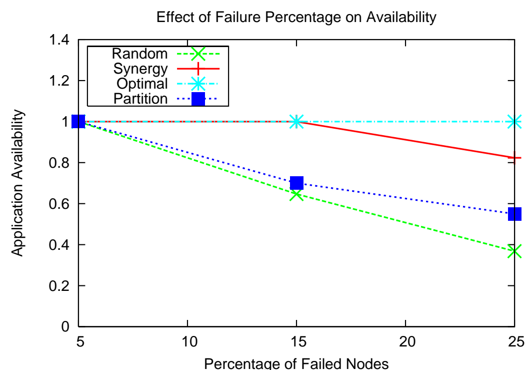

Figure 9: Failure percentage sensitivity.

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.

6.1 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 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 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 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.

Effect of scale. Figure 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 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.

6.2 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 be-

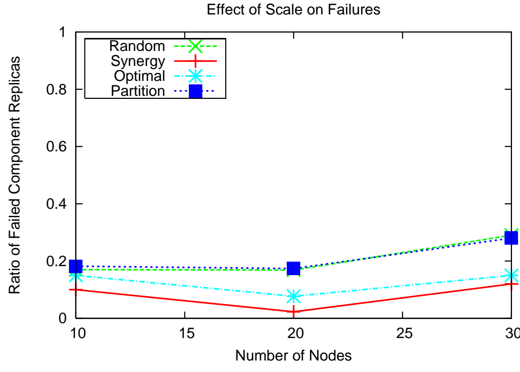

Figure 10: Failure ratio with scale.

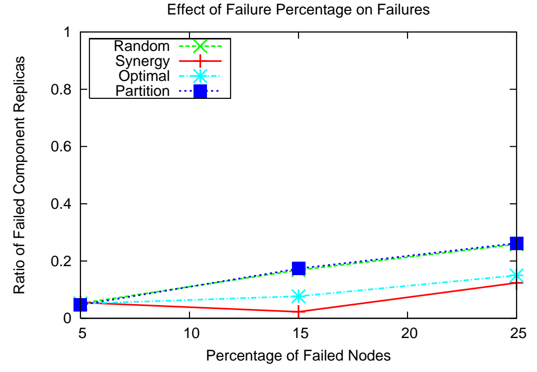

Figure 11: Failure ratio and percentage.

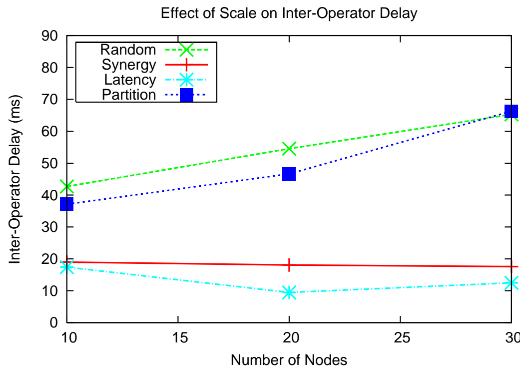

Figure 12: Scalability of inter-operator delay.

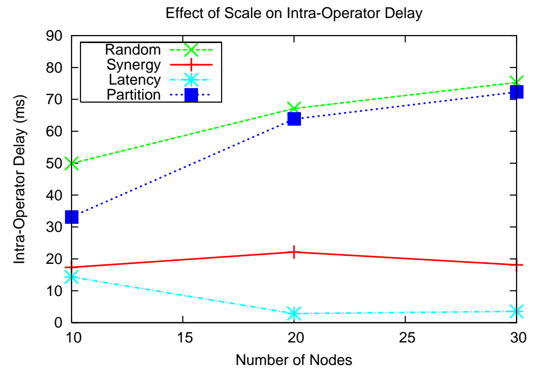

Figure 13: Scalability of intra-operator delay.

cause 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 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 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 11, are similar to the ones of the effect of network scale from Figure 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.

6.3 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 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 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 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.

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

Table 2: Latency information gathering.

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 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 2 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. 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.

7. RELATED WORK

Existing research in the area of high availability for distributed stream processing systems [5,10,15,16,32] has focused on efficient replica state maintenance to mask component failures. To this extent, recovery mechanisms [15], failure masking [32], consistency trade-offs [5], and checkpoint scheduling [10, 16] 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 [3, 23]. 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 [3], or employ a latency space [23]. 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 4.1, 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 [2, 4, 17, 30], distributed databases [13, 22, 24, 34], distributed object systems [11, 12, 14, 18, 21, 25, 36], and web services [6, 20] 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 [11, 12, 14, 21] or Enterprise JavaBeans [36], or on multi-tier architectures [13, 22] 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 [18] 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 [18] 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 [39] (with the theoretical analysis provided in [38]), 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 [39], 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.

8. CONCLUSION

In this paper we have studied the problem of component replica placement to achieve high availability in distributed stream processing applications. 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 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.

This is the first work we are aware of to discuss component replica placement for high availability in distributed stream processing systems. Our future work includes incorporating 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. Another area of future work includes the integration of our replica placement protocol with performance-oriented placement protocols, which includes maximizing the availability of already deployed application component graphs.

9. REFERENCES

[1] D. Abadi et al. The design of the Borealis stream processing engine. In *Proceedings of 2nd Biennial Conference on Innovative Data Systems Research, CIDR, Asilomar, CA, USA*, January 2005.

[2] 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.

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

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

[5] 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.

[6] 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*, September 2005.

[7] A. Bavier et al. Operating systems support for planetary-scale network services. In *Proceedings of 1st Symposium on Networked Systems Design and Implementation, NSDI, San Francisco, USA*, March 2004.

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

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] J. 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.

[16] J. 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.

[17] A. 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.

[18] S. Krishnamurthy, W. 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.

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

[20] 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.

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

[22] 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.

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

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

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

[26] T. Repantis, X. Gu, and V. Kalogeraki. Synergy: Sharing-aware component composition for distributed stream processing systems. In *Proceedings of 7th Middleware, Melbourne, Australia*, November 2006.

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

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

[29] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of 1st Middleware, Heidelberg, Germany*, November 2001.

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

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

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

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

[34] 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.

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

[36] 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.

[37] K.-L. Wu et al. Challenges and experience in prototyping a multi-modal stream analytic and monitoring application on system s. In *Proceedings of 33rd International Conference on Very Large Data Bases, VLDB, Vienna*, September 2007.

[38] H. Yu and P. 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.

[39] H. Yu, P. 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.