Adaptive Overlays for Shared Stream Processing Environments

Olga Papaemmanouil, Sujoy Basu, Sujata Banerjee

HP Laboratories
HPL-2007-178(R.1)

Keyword(s):
stream processing, overlay, quality-of-service

Abstract:
Large-scale overlays has become a powerful paradigm for deploying stream processing applications in wide-area environments. The dynamic nature of these systems makes it difficult to guarantee the Quality of Service (QoS) requirements of each application. In this work we present a framework for distributing stream processing applications, where processing components and stream flows could be shared across multiple applications. In our approach, nodes coordinate to precompute alternative network deployments for each application that respect both node constraints and the applications' QoS requirements. Given this set of alternative deployments, nodes can react fast to changes on the network conditions, workload or application expectation.

External Posting Date: October 6, 2008 [Fulltext] Approved for External Publication
Internal Posting Date: October 6, 2008 [Fulltext]
A Brief version published in Proceedings of the 4th International Workshop on Networking Meets Databases (NetDB)

Adaptive Overlays for Shared Stream Processing Environments

Olga Papaemmanouil
Brown University
olga@cs.brown.edu

Sujoy Basu
HP Labs
basus@hpl.hp.com

Sujata Banerjee
HP Labs
sujata@hpl.hp.com

ABSTRACT

Large-scale overlays has become a powerful paradigm for deploying stream processing applications in wide-area environments. The dynamic nature of these systems makes it difficult to guarantee the Quality of Service (QoS) requirements of each application. In this work we present a framework for distributing stream processing applications, where processing components and stream flows could be shared across multiple applications. In our approach, nodes coordinate to precompute alternative network deployments for each application that respect both node constraints and the applications' QoS requirements. Given this set of alternative deployments, nodes can react fast to changes on the network conditions, workload or application expectation.

1. INTRODUCTION

Over the part few years, *stream processing systems* (SPS) has gained considerable attention in a wide range of applications including planetary-scale sensor networks or "macroscopes" [3, 10], network performance and security monitoring [1, 2], multi-player online games and feed-based information mash-ups [4]. These applications are characterized by a large number of geographically dispersed entities: sources that generate potentially large volumes of data streams and consumers that register large number of concurrent queries over these data streams. To facilitate these applications, SPS systems need to provide high network and workload scalability. The former refers to the ability to gracefully deal with increasing geographical distribution of system components, whereas the latter addresses large number of simultaneous user queries. To achieve both types of scalability, the system should scale out and distribute its processing across multiple nodes, constituting distributed stream management systems [6, 17, 20].

While distributed versions of stream processing systems [6, 17, 20] and solutions for in-network stream processing [8, 25, 28] have been proposed, our work addresses the problem of network deployment of stream-based queries in shared processing environments. Sharing of data streams and processing components have been shown [22, 27] to improve resource utilization since redundant computations can be avoided. Moreover, shared data streams are forwarded to a single location for further processing, reducing network traffic. Thus, the processing capacity of the system can be improved by reusing previously generated data streams and computational results.

Existing work on shared-aware processing [22, 27] focuses on composing stream processing queries that reutilize existing processing components. However, the problem of in-network deployment of applications with shared components adds new challenges. First, applications often express Quality-of-Service (QoS) specifications, which describe the relationship between various characteristics of the output and its usefulness, e.g., utility, response delay, end-to-end loss rate, etc. For example, in many real-time financial applications, query answers are useful if they arrive timely. When applications are executed across multiple machines their QoS is affected by the location of each of their processing components. Thus, if some of these components are shared across multiple queries, their location will have an impact on the service level of all their dependent applications.

Moreover, stream processing applications are expected to operate over the public Internet, with large number of unreliable receptors, on commodity machines, some or all of which may contribute their resources only on a transient basis (e.g., as in the case of peer-to-peer settings). In general, distributed steam processing systems should be able to deal with churn, time-varying workload and resource availability. Thus, in-network deployments of processing queries should be periodically reevaluated to adapt to any dynamic changes. In shared environments, periodic (or occasionally concurrent) modifications of the current deployments (through migration or replication of the processing components) require special attention, as multiple applications, with possible conflicting QoS expectations, could be affected.

This paper addresses the aforementioned challenges of shared stream processing systems. We propose an adaptive overlay-based framework that distributes stream processing queries across multiple available nodes. In our system, nodes self-organize on a distributed resource directory service, which they can use for advertising and discovering available resources. Our framework strives to identify deployments of multiple shared stream processing queries that respect resource constraints of the network nodes and QoS expectations of each application, while maintaining a low bandwidth consumption. As opposed to previous work [8, 25, 28], we follow a proactive approach, where nodes periodically collaborate to *precompute* alternative deployments of the registered queries. During run time, when QoS violations occur, the system can react fast to changes and migrate to a valid state (i.e., with no violations) by applying the most suitable of the precomputed plans. Moreover, even in the absence of any violations, the best of these plans can be applied periodically in order to improve the bandwidth

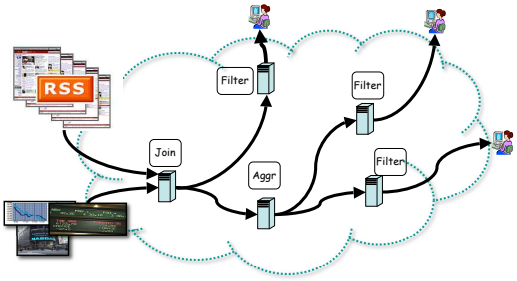

Figure 1: Query deployment example.

consumption of the system.

2. SYSTEM MODEL

Our system consists of a large number of *brokers* (or nodes), providing stream-based query processing services [5, 24]. Data sources (or publishers) publish data streams while clients (or subscribers) subscribe their data interests expressed as stream-oriented continuous queries. The system streams data from publishers to subscribers via processing operators deployed on the network, an example of which is shown in Figure 1. In the rest of the discussion, we use the terms node and broker interchangeably.

Stream Processing Model. Client queries are defined as directed, acyclic data-flow graphs of stream-oriented operators [5, 24]. Multiple queries can be expressed, each interconnecting multiple operators. We assume that each processing operator has specific requirements, i.e., CPU, memory, disk space, etc, which can be required through resource profiling techniques [7]. In our prototype, we focus on CPU requirements, measured in CPU cycles per stream tuple. We refer to this as the cost of an operator o_i, oc_{o_i}. Moreover, given the input rate of each operator, $r^{in}_{o_i}$ in tuples per time period, the load of each operator (in CPU cycles per time period) is defined by the product $oc_{o_i} \times r^{in}_{o_i}$.

Clients can also express their QoS expectations for the queries they register to the system, such as end-to-end latency, loss rate and available bandwidth. In this work, we have used end-to-end latency as the QoS metric. Thus, each query q_i is associated with a maximum end-to-end latency expectation QoS_{q_i}. Without loss of generality, latency can be replaced with another additive metric.

Processing Overlay. Operators are free to roam in the network and may be reallocated over time as part of our optimization process. The location of a query's operators define the *deployment plan* of the query. Depending on the resources available in the network and the query's requirements, each query could have multiple alternative deployment plans. The operators of a query are interconnected by *overlay links*, each forwarding the output of an operator to the next processing operator in the query plan. Thus, query deployments create an overlay network with a topology consistent with the data flow of the registered queries.

If an operator o_i forwards its output to o_j we refer to o_i as the *upstream* operator of o_j (or its publisher) and to o_j as the downstream operator of o_i (or its subscriber). Operators could have multiple publishers (e.g., join, union) and since they could be shared across queries they could also have multiple subscribers. We denote the set of subscribers of o_i as sub_{o_i} and its set of publishers as pub_{o_i}.

Resource Directory Overlay. To discover potential hosts of the processing operators, we rely on a distributed resource discovery service, implemented and maintained by the nodes in our system. Our implementation is based on the NodeWiz [9] system, a scalable tree-based overlay infrastructure for resource discovery. Nodes can use NodeWiz to advertise the attributes of available resources and efficiently perform multi-attribute queries to discover the advertised resources. NodeWiz can adapt the assignment of the multi-attribute space to the nodes such that the load of distributing adverts and performing queries is balanced across nodes.

In our model, we assume that each node n_i has a CPU capacity c_i. Distributing operators in the network affects the residual capacity of the available nodes. More specifically, if O_{n_i} is the set of operator executed on node n_i, then the residual CPU capacity of each node is

$$c_i - \sum_{o_j \in O_{n_i}} oc_{o_j} \times r^{in}_{o_j}$$

Nodes periodically advertise their residual capacity to NodeWiz and query the directory to discover nodes that have enough CPU capacity to host and execute a specific operator. The queries are issued when needed as part of our proactive operator placement approach.

Network Monitoring Service. Our overlay-based middleware relies on a network monitoring service for collecting latency statistics of the overlay links between nodes in our system. In our implementation we use S^3 [33], a scalable sensing service for real-time and configurable monitoring for large networked systems. The infrastructure measures both network and node metrics, while it aggregates data in a scalable manner. Moreover, inference algorithms can be used to derive path properties of all pairs of nodes based on a small set of network paths. S^3 is currently deployed on PlanetLab and performs measurements of several network metrics.

3. OPTIMIZATION

Our optimization framework distributes processing operators, aiming to meet the QoS expectations of each query and respect the resource constraints of the nodes. In the rest of the section, we describe our approach in detail.

Optimization goal. Across all possible deployments of the registered queries, only a subset of them are *feasible*. The following definition describes these plans.

DEFINITION 1. *Given a set of nodes N with CPU capacities $c_i, \forall n_i \in N$ and a set of operators O shared across a query set Q, with QoS expectations $QoS_{q_t}, \forall q_t \in Q$, a feasible deployment plan of the operator set O across the set of brokers N, is one that satisfies the following:*

$$\sum_{o_j \in O_{n_i}} oc_{o_j} \times r^{in}_{o_j} \leq c_i, \forall n_i \in N$$

$$d_{q_t} \leq QoS_{q_t}, \forall q_t \in Q$$

where the function $h(o_i) : O \rightarrow N$ provides the host node of o_i. The latency of a query q_t, d_{q_t}, is the end-to-end response latency of the query, measured by aggregating the network latency of all overlay links connecting the operators of the query to its publisher (or a source). If an operator has multiple publishers, then the overlay path with the maximum latency is used.

In this work, we assume that we maintain CPU loads at each node under a certain threshold, i.e., the residual CPU capacity will never be planned to be zero. Under this assumption, processing delays are negligible compared to network delays, thus, $d(n_i, n_j)$ refers to the network latency of the overlay link between n_i and n_j. Given a set of feasible plans our goal is to deploy the one that minimizes the total bandwidth consumption, i.e., the sum of outgoing data rate over all nodes in our system. Formally, this is defined as:

$$\min \sum_{n_i \in N} b_i$$

$$b_i = \sum_{o_j \in O_{n_i}} \sum_{o_m \in sub_{o_j}} r^{in}_{o_j} \times s_{o_j} \times \phi(h(o_j), h(o_m))$$

$$\phi(v,u) = \begin{cases} 1 & v \neq u \\ 0 & \text{otherwise} \end{cases}$$

where s_{o_i} is the selectivity of the operator o_i.

3.1 Design Goals

Our framework strives to meet three requirements. First, it should be *adaptive*, even to load spikes, and continuously maintain an efficient network deployment of each query. Any QoS or resource violations should be detected and addressed as soon as possible. To achieve this, our framework must be light-weight and the deployment plans should be efficiently discovered. Second, our protocol should be *scalable* in terms of the network size, the number of queries and the degree of sharing across their processing operators. Finally, since operator migrations could affect multiple queries, the solution should be equally attentive to each individual operator's placement and query's QoS.

To address these challenges, we designed a decentralized optimization framework that does not rely on global information and has low communication cost. Our protocol periodically *precomputes* alternative feasible deployment plans for all registered queries. Each broker maintains information regarding the placement of its local operators and periodically collaborates with nodes in its "close neighborhood" to compose deployment plans that distribute the total set of query operators. Whenever a resource or QoS violation occurs, the system can react fast by applying the most suitable plan from the precomputed set. Moreover, even in the absence of violations, the system can periodically improve its current state by applying a more efficient deployment than the current one. Finally, our protocol includes a replication-based approach for handling conflicting concurrent modifications of queries.

3.2 Approach overview

We propose a proactive distributed operator placement algorithm which is based on informing downstream operators about the feasible placements of their upstream operators. This way nodes can take decisions regarding the placement of their local and upstream operators that will influence their shared queries the best way possible. The main advantage of our approach is that nodes can make placement decisions on their own, which provides fast reaction to any QoS violations.

Each operator periodically sends deployment plans to its subscribed downstream operators describing possible placements of their upstream operators. We refer to these plans as *partial*, since they only deploy a subset of a query's operators. When a node receives a partial plans from the publisher of local operator it extends the plan by adding the possible placements of the local operator. Partial plans that meet the QoS expectations of all queries sharing the operators in the plan are propagated. To identify feasible deployments we employ a search algorithm, which we call *k-ahead search*. The algorithm discovers the placement of k operators ahead from the local operator that incurs the lowest latency. Based on this minimum latency, we eliminate, as early in the optimization process as possible, partial plans that could violated any QoS expectations.

Finally, every node *finalizes* its local partial plans: for each one of them, it evaluates its impact on the bandwidth consumption and the latency of all affected queries. Using the final plans, a node can make fast placement decisions in run-time. Next we describe these steps in detail.

Symbol	Definition
oc_{o_i}	cost of operator o_i
$r^{in}_{o_i}$	input rate of operator o_i
QoS_{q_t}	QoS of query q_t
d_{q_t}	response latency of query q_t
sub_{o_i}	subscribers (downstream operators) of o_i
pub_{o_i}	publishers (upstream operators) of o_i
$h(o_i)$	host node of operator o_i
c_i	capacity of node n_i
O_{n_i}	set of operators hosted on n_i
Q_{o_i}	set of queries sharing operator o_i
A_{o_i}	candidate hosts of operator o_i
P_{o_i}	upstream operators of o_i
$O(q_i)$	set of operators in query q_i

Table 1: Notation.

3.3 Deployment Plan Generation

Each node periodically identifies a set of partial deployment plans for all its local operators. Let us assume an operator o_i shared by a set of queries $q_t \in Q_{o_i}$. Let also P_{o_i} be the set of upstream operators for o_i. An example is shown in Figure 2. Queries q_1 and q_2 share operators o_1 and o_2 and $P_{o_3} = P_{o_4} = \{o_1, o_2\}$.

A partial deployment plan for o_i assigns each operator $o_j \in P_{o_i} \cup \{o_i\}$ to one of the nodes in the network. Each partial plan p is associated with (a) a *partial cost*, pc^p, i.e., the bandwidth consumption it occurs, and (b) a *partial latency* for each query it affects, $pl^p_{q_t}, \forall q_t \in Q_{o_i}$. For example, a partial plan for o_2 will assign operators o_1 and o_2 to two nodes, evaluate the bandwidth consumed due to these placements, and the response latency up to operator o_2 for each query q_1 and q_2.

In the rest of the section, we describe how deployment plans are generated. We start with the k-ahead search algorithm.

3.3.1 k-ahead search

Every node n_v runs the k-ahead search for each local operator $o_i \in O_{n_v}$ and each candidate host for that operator. If A_{o_i} is the set of candidate hosts for o_i, the search identifies the minimum latency placement of k operators ahead of o_i for each of the queries sharing o_i, assuming that o_i is placed on the node $n_j \in A_{o_i}$. Intuitively, it attempts to identify the minimum impact on the latency of each query $q_t \in Q_{o_i}$, if we migrate o_i to node n_j and make the best placement decision (wrt latency) for the next k downstream

operators of each query q_t. Below we describe the steps of the algorithm, which initially evaluates the 1-ahead latency and then derives the k-ahead latency value for every triplet (o_i, n_j, q_t), where $o_i \in O_{n_v}, n_j \in A_{o_i}, q_t \in Q_{o_i}$.

For each operator $o_i \in O_{n_v}$, n_v executes the following steps:

1. Identifies the candidate hosts A_{o_i} of the local operator o_i by querying the resource directory service. Assuming the constraint requirements of o_i are $C = [(c_1, v_1), (c_2, v_2), ..., (c_m, v_m)]$, where c_i is the resource attribute and v_i is the operator's requirement for that resource, we query the resource directory for nodes with
$$c_1 \geq v_1 \wedge c_2 \geq v_2 \wedge ... \wedge c_m \geq v_m.$$

2. If o_m is the downstream operator of o_i for the query $q_t \in Q_{o_i}$, the node sends a request to the host of o_m, asking for the set of candidate hosts A_{o_m} of that operator. For each one of these candidate nodes, it queries the networking monitoring service for the latency $d(n_j, n_t), \forall n_j \in A_{o_i}, \forall n_t \in A_{o_m}$. The 1-ahead latency for the o_i operator with respect to its candidate n_j and the query $q_t \in Q_{o_i}$ is
$$\gamma_i^1(n_j, q_t) = \min_{n_u \in A_{o_m}} \{d(n_j, n_u)\}$$
In Figure 2, $sub_{o_2}^{q_1} = o_4$, $sub_{o_2}^{q_1} = o_3$ and n_1 will request from n_2 the candidate hosts A_{o_2} for the operator o_2, and will estimate the 1-ahead latencies $\gamma_1^1(n_5, q_1) = \gamma_1^1(n_5, q_2) = 10ms$. Also for o_2 we assume $\gamma_2^1(n_6, q_1) = 5ms$ and $\gamma_2^1(n_6, q_2) = 15ms$.

3. The algorithm continues in rounds, where for each operator o_i the node waits for it subscribers o_m in the query $q_t \in Q_{o_i}$ to complete the evaluation of the $(k-1)$-ahead latency before they proceed with the estimation of the k-ahead latency. The k-ahead latency for the o_i operator with respect to its candidate n_j and the query $q_t \in Q_{o_i}$ is
$$\gamma_i^k(n_j, q_t) = \min_{n_u \in A_{o_m}} \{\gamma_i^{k-1}(n_t, q_t) + d(n_j, n_u)\}$$

Due to space constraints, we omit the detailed algorithm for the k-ahead search and illustrate the last step using the example in Figure 2. In this case, $\gamma_1^2(n_5, q_1) = \min\{(10 + \gamma_2^1(n_6, q_1), 30 + \gamma_2^1(n_9, q_1)\} = 15ms$, and $\gamma_1^2(n_5, q_2) = \min\{10 + \gamma_2^1(n_6, q_2), 30 + \gamma_2^1(n_9, q_2)\} = 25ms$. Thus, assuming we migrate o_1 to n_5, the placement with the minimum latency of the next two operators will increase the response latency of q_1 by 15ms and the latency of q_2 by 25ms.

3.3.2 Plan generation

Our plan generation process is initiated by nodes executing the "leaf" operators of a query, i.e., operators that receive streams from the sources. We first describe how partial deployment plans are created at the leaf nodes and then we focus on the nodes executing intermediate operators. We assume that each node is aware of the candidate hosts of each local operator, as the k-ahead search algorithm precedes the plan generation process.

Leaf nodes. Let o_i be a leaf operator executed on a node n_v. Node n_v will create a set of partial plans, each one assigning o_i to a different candidate host $n_j \in A_{o_i}$ and evaluate its partial cost and the partial latencies of all queries sharing o_i. If S_{o_i} is the set of input sources for o_i, and $h(s), s \in S_{o_j}$, is the node publishing data on behalf of source s, then, the partial latency (i.e., the latency from the sources to the n_j) of a query q_t is
$$pl_{q_t}^p = \max_{s \in S_{o_i}} d(h(s), n_j), \forall q_t \in Q_{o_i}.$$
Finally, since this plan assigns the first operator, its partial bandwidth consumption is zero.

Plan Elimination. Once a partial plan is created, we need to decide whether we should forward it downstream and expand it by adding more operator migrations. A partial plan is propagated only if it could lead to a feasible deployment. The decision is based on the results of the k-ahead search. The k-ahead latency for a triplet (o_i, n_j, q_t) represents the minimum latency overhead for a query q_t across all possible placements of k operators ahead of o_i, assuming o_i is placed on n_j. If the latency of the query up to operator o_i plus the minimum latency for k operators ahead violates the QoS of the query, the partial plan could not lead to any feasible deployments. More specifically, a partial plan p that places operator o_i to node n_j is infeasible if there exists at least one query $q_t \in Q_{o_i}$ such that,
$$pl_{q_t}^p + \gamma_i^k(n_j, q_t) \geq QoS_{q_t}.$$

Note, that the k-ahead latency, although it does not eliminate feasible plans, it does not identify *all* infeasible deployments. Thus, the propagated plans are "potentially" feasible plans which may be proven infeasible in following steps. Moreover, there is a tradeoff with respect to the parameter k. The more operators ahead we search, the higher the overhead of the k-ahead search, however, the earlier we will be able to discover infeasible plans. Our initial experiments reveal that our approach manages to eliminate a significant number of redundant partial plans early in the propagation process, keeping the optimization traffic small.

Intermediate nodes. Partial plans that are not eliminated are forwarded downstream, along with their metadata (i.e., partial cost and latencies). Let us assume a node n_v, processing an operator o_i, receives a partial plan p from its publishers $o_m \in pub_{o_i}$. For purposes of illustration we assume a single publisher. Our equations can be generalized for multiple publisher in a straightforward way. Note, that each query sharing o_i is also sharing its publishers. Thus, each received plan includes a partial latency $pl_{q_t}^p, \forall q_t \in Q_{o_i}$. Our protocol expands each of these plans by adding migrations of the local operator o_i to its candidate hosts.

For each candidate host $n_j \in A_{o_i}$, the node n_v validates the resource availability: it parses the plan p to check if any upstream operators have also been assigned to n_j. To facilitate this, we send along with each plan metadata on the expected load requirements of each operator included in each plan. If the residual capacity of n_j is enough to process all assigned operators including o_i, we estimate the impact of the new partial plan f:
$$pl_{q_t}^f = pl_{q_t}^p + d(h^p(o_m), n_j), \forall q_t \in Q_{o_i}$$
$$pc^f = pc^m + r_{o_m}^{out} \times \phi(h^p(o_m), n_j)$$
where, $h^p(o_m)$ is the host of o_m in the partial plan p. For each new partial plan f we also check if it could lead to a feasible deployment, based on the k-ahead latency $\gamma_i^k(n_j, q_t)$, and propagate only feasible partial plans.

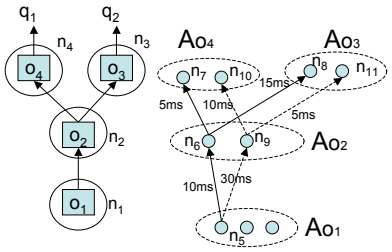

Figure 2: Query example.

Final plan generation. Partial plans created on a node are "finalized" and stored locally. To finalize a partial plan we quantify its impact on the current bandwidth consumption and on the latency of the queries it affects. To implement this process, we maintain statistics on the bandwidth consumed by the upstream operators of every local operator and the query latency up to this local operator. For example, in Figure 2, if o_1 is a leaf operator, n_2 maintains statistics on the bandwidth consumption from o_1 to o_2 and the latency up to operator o_2. For each plan, we evaluate the difference of these metrics between the current deployment and the one suggested by the plan and store this metadata along with the corresponding final plan. Thus, every node stores a set of feasible deployments for its local and upstream operators, along with the effect of these deployments on the system cost and the latency of the queries. In Figure 2, n_2 stores plans that migrate operators $\{o_1, o_2\}$, while n_4 will store plans that place $\{o_1, o_2, o_4\}$.

Combining and expanding partial plans received from the upstream nodes may generate a large number of final plans. To deal with this problem, we employ a number of elimination heuristics. For example, among final plans with similar impact on the query latencies we keep the ones with the minimum bandwidth consumption, while if they have similar impact on the bandwidth we store the ones that reduce the query latency the most. We omit the rest of the heuristics due to space limitations.

3.4 Run-time optimization

In this section we describe how queries are initially deployed in the network and how we can use our precomputed deployment plans to adapt to dynamic changes in our system.

Initial placement. When a client registers a new query our system tries to identify if any of the operators and streams of the query already exist in the system. Discovery of processing components and data streams has been addressed in [22, 27] and it is outside the scope of this paper. In this work, we assume that all available components are advertised to our resource directory service using a global naming schema and are located by querying the directory. Operators that do not exist are assigned for execution to the node closest to their publisher operator with enough CPU capacity. Once, the query is deployed, we execute our plan generation algorithm and across all discovered plans we apply the one that provides the minimum bandwidth consumption. This plan will meet the latency expectation of the query without violating the QoS of the rest of the queries. If no feasible plan can be discovered the query is rejected.

Adaptive optimization. Our system periodically initiates the plan generation process and creates plans that reflect the most current workload and network conditions. If this process discovers query deployments that can improve the bandwidth consumption we apply the one with the best impact.

The precomputed deployments plans stored on our nodes are used whenever changes in the system violate the QoS of some queries. To detect these violations, every node monitors for every local operator the latency to the location of its publishers. It also periodically receives the latency of all queries sharing its local operators, and it quantifies their "slack" from their QoS expectations, i.e., the increase of latency each query can tolerate.

Let us assume an operator o_i with a single publisher o_m and shared by a query q_t with a response delay d_{q_t} and slack $slack_{q_t}$. If the latency of the overlay link between o_i and o_m increases by

$$\Delta d(h(o_m), h(o_i)) > slack_{q_t},$$

then the QoS of the query q_t is violated and a different deployment should be applied immediately.

Across all final plans stored at the host of o_i, we search for the a plan p that decreases q_t's latency by at least $\Delta pl^p_{q_t} = d_{q_t} - QoS_{q_t}$. Across all plans to satisfy this condition, we remove any plan p that does not migrate o_i and o_m (i.e., includes the bottleneck link) and satisfies:

$$\Delta pl^p_{q_t} + \Delta d(h(o_m), h(o_i)) \leq QoS_{q_t} - d_{q_t}.$$

From the remaining plans, we apply the one with that improves the most the bandwidth consumption.

If a deployment that can meet q_t's QoS can not be discovered at the host of o_i, the node sends a request for a suitable plan to its subscriber for the violated query q_t. The request includes also metadata regarding the congested link (e.g., its new latency). Nodes that receive such requests, attempt to discover (in the same way described above) a plan that can satisfy the QoS of the query q_t. Since downstream nodes store plans that migrate more operators, they are more likely to discover a feasible deployment for q_t. The propagation continues until we reach the node hosting the last operator of the violated query. If a feasible plan does not exist, then the query q_t could not be satisfied.

It is important to mention that identifying a new deployment has a small overhead. Essentially, nodes have to search for a plan that reduces enough the latency of a query. Final plans can be indexed based on the queries they affect and sorted based on their impact on each query's latency. Thus, when a QoS violation occurs, our system can identify its "recovery" deployments very fast.

3.5 Concurrent modifications

Concurrent modifications of shared queries require special attention, as they could create conflicts with respect to final latency of their affected queries. For example, in Figure 2, assume that the QoS of both q_1 and q_2 are not met, and nodes n_3 and n_4 decide concurrently to apply a different deployment plan for each query. Parallel execution of these plans does not guarantee that their QoS expectations will be satisfied.

To address the problem, we rely on *operator replication*. We apply deployment plans by replicating the operators whenever migrating them cannot satisfy the QoS constraints of *all* their dependent queries. However, replicating processing increases the bandwidth consumption as well as the processing load in our system. Hence, our protocol identifies if conflicts could be resolved by alternative candidate

plans, and if none is available, then it applies replication. Our framework exploits the metadata created during the plan generation phase to identify alternative to replication solutions. More specifically, it uses the existing deployment plans to

- decide whether applying a plan by migration satisfies all concurrently violated queries,
- allow multiple migrations whenever safe, i.e., allow for parallel migrations,
- build a non-conflicting plan when the existing ones can cannot be used.

In the next paragraph we describe our approach, starting with the following definitions.

DEFINITION 2. (DIRECT DEPENDENCIES) *Two queries q_i and q_j are* directly dependent *if they share an operator, i.e.,* $\exists o_k$ such that $q_i \in Q(o_k)$ and $q_j \in Q(o_k)$. *Then, q_i and q_j are* dependent queries *of every operator o_k. We note the set of dependent queries of a query q_i as D_{q_i} and the dependent queries of an operator o_k as D_{o_k}. Then, if $O(q_i)$ is the set of operators in query q_i, $D_{q_i} = \bigcup_{o_k \in O(q_i)} D_{o_k}$.*

Directly dependent queries do not have independent plans, therefore concurrent modifications of their deployment plans require special handling to avoid any conflicts and violation of the delay constraints.

DEFINITION 3. (INDIRECT DEPENDENCIES) *Two queries q_i and q_j are* indirectly dependent *iff $O(q_i \cap q_j) = \oslash$ and $D_{o_i} \cap D_{o_j} \neq \oslash$.*

Indirectly dependent queries have independent (non-overlapping) plans. Nevertheless, concurrent modifications on their deployment plans could affect their common dependent queries. Hence, our framework should address these conflicts as well, insuring that the QoS expectations of the dependent queries are satisfied.

To detect concurrent modifications, we follow a lease-based approach. Once a node decides that a new deployment should be applied, all operators in the plan and their upstream operators are locked. Nodes trying to migrate already locked operators check if their modification does not conflict with the current one in progress. If a conflict exists, it tries to identify an alternative non-conflicting deployment. Otherwise, it applies its initial plan by replicating the operators. In the next paragraph, we describe in more detail our solution.

Lease-based algorithm. Assume a node has decided on the plan p to apply for a query q. It forwards a $REQUEST\_LOCK(q, p)$ message to its publishers and subscribers. In order to handle indirect dependencies, each node that receives the lock request, will also send it to the subscribers of its local operator of the query q. This request informs nodes executing any query operators and their dependents about the new deployment plan and request the lock of q and its dependents. Given that no query has the lock (which is always true for queries with no dependents), publishers/subscribers reply with a $MIGR\_LEASE(q)$, once they receive a $MIGR\_LEASE(q)$ from their own publisher/subscriber of that query. Nodes that have granted a *migration lease* are not allowed to grand another migration lease until the lease has been released (or expired, based to some expiration threshold).

Once node n receives its migration lease from all its publishers and subscribers of q, is applies the plan p for that query. It will parse the deployment plan and for every node hosting a migrating operator o to node n sends a $MIGRATE(o, n)$ message. Migration is applied in a top-down direction of the query plan, i.e. the most upstream nodes migrate their operator (if required by the plan) and once this process is completed the immediate operators are informed about the change and subscribe to the new location of the operators. As nodes update their connections, they apply also any local migration specified by the plan. Once the whole plan is deployed then a $RELEASE\_LOCK(q)$ request is forwarded to the old locations of the operators and their dependents, which release the lock for the query.

A lock request is send across all nodes hosting operators included in the plan and all queries sharing operators of the plan. Once the lock has been granted any following lock requests will be satisfied either by replication or migration lease. A migration lease allows the deployment plan to be applied by migrating its operators. However, if such a lease cannot be granted due to concurrent modifications on the query network, a *replication lease* can be granted, allowing the node to apply the deployment plan of that query by replicating the involved operators. This way, only this specific query will be affected.

PROPERTY 1. *If an operator o_i is shared by a set of queries D_{o_i}, then the sub-plan rooted from o_i is also shared by the same set of queries.*

Let us now assume two dependent queries q_i and q_j that both have their QoS violated. Query q_i sends the $REQUEST_LOCK(q_i, p_i)$ requests to this downstream operators and similarly for the query q_j. Moreover, shared operators that are aware of the dependencies, they forward also the same request to their subscribers, to inform also the dependent queries of the requested lock. Since queries share some operators, at least one operator will receive both lock requests. Upon receipt of the first requests it applies the procedure describe below, i.e., identifying conflicts and resolving them based on the metadata of the two plans. However, when the second request for a lock arrives the first shared node to receive does not forward it to any publishers as a migration lease for this query has already been granted. In the rest of the section, we describe the different cases we encounter when trying to resolve conflicts.

Direct dependencies.

In this section we describe how we handle concurrent modifications on directly dependents plans. More specifically, we describe in which cases we can avoid granting a replications lease.

Parallel migrations. Concurrent modifications are not always conflicting. If two deployment plans do not affected the same set of queries, then both plans can be applied in parallel. For example, in Figure 2, if n_3 and n_4 decide to migrate only o_3 and o_4 respectively, they both changes can be applied. In this case, the two plans decided by n_3 and n_4 should show no impact on the queries q_1 and q_2 respectively. Our deployment plans include all the necessary information (operators to be migrated, new hosts, affect on the queries)

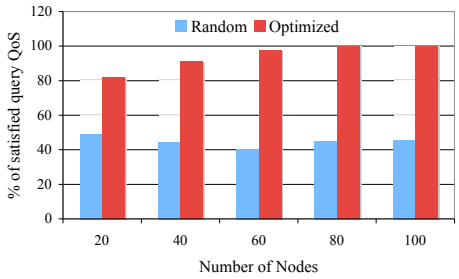

Figure 3: QoS expectations satisfied for different network sizes.

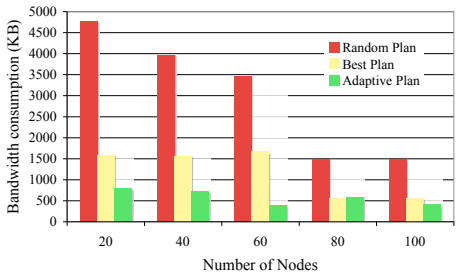

Figure 4: Total bandwidth consumption for different network sizes.

to identify these cases efficiently an thus grant migration leases to multiple non-conflicting plans.

Redundant migrations. Multiples migrations defined by concurrent deployment of multiple plans may often not be necessary in order to guarantee the QoS expectations of the queries. Very often, node might identify in parallel QoS violations and attempt to address them by applying their own locally stored deployment plans. In this case, it is quite possible that either one of the plans will be sufficient in order to reconfigure the current deployment. In our approach, every plans includes an evaluation of the impact on all affected queries. Thus, if two plans p_1 and p_2 are both affecting the same set of queries, then applying either one will still provide a feasible deployment of our queries. Therefore, our system will apply the plan that will first acquire the migration lease while the second plan will be ignored.

Alternative migration plan. Deployments plans that relocate shared operators can not be applied in parallel. In this case, the first plan to request the lock migrates the operators, while we attempt to identify a new alternative non-conflicting deployment to meet any unsatisfied QoS expectations. Since the first plan is migrating a shared operator, then we look in the hosts of its downstream operators to discover any plans that were built on top of this migration. For example, in Figure 2, if the first plan migrates operator o_1, but the QoS of q_2 is still not met, we search in node n_4 for any plans that include the same migration for o_1 and can reduce further q_2's response delay, i.e., by migrating o_4 as well.

Indirect dependencies.

As mentioned above, queries may not share operators, but still share dependents. Thus, if we attempt to modify the deployment of indirectly dependent queries, we need to take under consideration the impact on their shared dependents. In this case, we grant a migration lease to the first lock request and a replication lease to any following requests, iff the plans to be applied are affecting overlapping sets of dependent queries. However, in the case of that they do not affect the QoS of the same queries, these plans can be applied in parallel.

4. PRELIMINARY EVALUATION

We developed an initial prototype in Java and run some preliminary experiments distributing up to 500 queries to 100 nodes. The nodes are located in a local data-center, however the latencies between nodes represent PlanetLab sites' pair-wise latencies, obtained from the PlanetLab deployment of S^3 [33]. All nodes are organized in the NodeWiz overlay, where they advertise their CPU availability.

Our queries are composed by a set of feed-based processing operators, similar to the operators used in Yahoo!Pipes [4], a centralized feed manipulation web service. The operators can union, filter, split, or sort RSS feeds. In our experiments, each query is a chain of up to h operators, and we set h to five. To create shared queries, we treat each operator as a separate query, thus, if an operator is in the i-level in the chain then it is shared by $(h - i + 1)$ queries. The input streams of each query are randomly chosen from a set of 100 RSS feeds published from an RSS source. The average feed size of 6.3KB. RSS sources are assigned to a random set of brokers which publish their feeds. Clients are also hosted by a random node.

Figure 3 shows the percentage of queries that meet their QoS expectations for different network sizes for two different deployment approaches. *Random* assigns each operator to a random node with enough CPU to process the operator. *Optimized* uses our framework to discover feasible deployments and applies for each query chain the one with the minimum bandwidth consumption. The results show that our approach improves the number of queries that meet their QoS expectations by 39%-58% . Moreover, as the network size increases and more resources become available, we are able to satisfy all queries. Our experiments also revealed that the average response latency of the queries decreases, compared with the Random deployment, by 45%-52%, depending on the network size and workload. We omit the figures due to space limitations.

To test the adaptivity of our system, we used the following approach. We use our framework to acquire a set of feasible query deployment plans and for each query chain we initially apply a random plan (*Random Plan*). Note, that this is a different deployment than the previous Random, as it takes also QoS expectations into consideration and tries to meet them. Once all operators are placed in the network, our system can identify better feasible plans, and gradually changes the deployment to meet these best plans, achieving the bandwidth consumption shown by *Best Plan*. At this point, we reduce the input rate of our sources by half and expect our system to adapt to the change by applying a different deployment (*Adaptive Plan*). Figure 4 shows the results. The best plan can achieve a significant improvement over a random feasible deployment. Moreover, when we decrease the input rate, we decrease the workload of our nodes and we discover plans that place chains of operators on the same nodes, reducing in-network stream forwarding. Finally, as we increase our network size, we make more resources available, thus, more operators can be processed in a single node, decreasing even more the bandwidth consumption.

5. RELATED WORK

Distributed query processing. Distributed query optimization and in particular the site selection problem are closely related to our work and have been explored extensively in the context of distributed and federated databases [14, 15, 18, 23, 30]. To the best of our knowledge, none of these approaches address widely-distributed processing and network awareness.

More recent work addressed Internet-scale query processing and distribution scalability. IrisNet [10] focuses on querying wide-area sensor databases using XPath queries. IrisNet relies on the DNS to identify the remote databases relevant to a given query, which is then processed using XML and XPath specific optimizations. Similar to our work, PIER [16] addresses DHT-based highly-distributed query processing, although in a pull-based setting. PIER discusses how CAN [26] can be used as a hashing function on the indexes of relations, distributing tuples across a very large number of sites. Our algorithms attempt a finer-grained control of the placement decisions, whereas in PIER, the operations themselves are randomly distributed across peers by CAN. The semantic details of our operators are abstracted away from the placement mechanism. Instead we focus on optimizing the network positioning of operators, and as such, operator specific optimizations such as those presented by PIER may still apply.

Distributed stream processing. Recently, there have been a number of efforts focusing on stream processing [5, 24] and frameworks for in-network deployment of continuous queries [6, 11, 17]. Borealis [6] is a distributed stream-based processing system. that inherits core stream processing functionality from Aurora [5] and distribution capabilities from Medusa [10]. Borealis includes a optimization framework that includes three levels of collaborating optimizers. At the lowest level, a local optimizer runs at every site and is responsible for scheduling messages to be processed as well as deciding where in the locally running diagram to shed load, if required. A neighborhood optimizer also runs at every site and is primarily responsible for load balancing the resources at a site with those of its immediate neighbors. At the highest level, a global optimizer is responsible for accepting information from the end-point monitors and making global optimization decisions.

GATES [11] also provides a middleware for processing of distributed data streams. This system is designed to use the existing grid standards and tools to the extent possible and uses a self-adaptation algorithm that achieves the best accuracy that is possible while maintaining the real-time constraint on the analysis.

Stream Processing Core (SPC) [17], on the other hand, was designed to address large scale (i.e., leveraging potentially thousands of computational nodes) distributed stream mining applications. It is designed with the assumption that the system is constantly overloaded with respect to the available resources. For this reason, SPC has to use resources intelligently in order to minimize the loss of useful data. A key distinguishing feature of SPC is dynamic application composition which enables stream connections to be made and broken dynamically as new applications and new data sources join and leave the system.

Borealis [6] supports run-time operator migration (but not operator replication), yet Borealis, GATES and SPC currently avoid the operator placement problem by supporting only pre-defined operator locations with pinned operators in the network. This leaves the burden of efficient operator placement to the system administrator, which is infeasible for a dynamic, large-scale system with thousands of queries.

Operator placement. The operator placement problem has been studied in [8, 21, 25, 27]. SAND [8], proposes a set of approaches or in-network placement of stream processing operators. Operators are placed either at the consumer side, at the producer side, or in-network on a DHT routing path between the two endpoints, depending on the bandwidth usage of a query. Applications can also specify delay constraints on the placement path in the DHT. Our approach of performing operator placement is more general than SAND because placements are not tied to DHT routing paths or to a specific optimization metric. Moreover, previous work [42] has shown that DHT routing paths can lead to inefficient candidate sets for operator placement. This is because DHT routing tables are optimized for minimizing hop count and not for delay or bandwidth usage.

IFLOW [21, 20, 19] propose a resource-aware approach to distributed stream management. The approach makes use of in-network data aggregation to distribute the processing and reduce the communication overhead involved in large scale distributed data management. Moreover, the system support for high-level language constructs to describe data-flows and deploys these data-flows across the network and reconfiguration the deployment in response to change in operating conditions. Their solution does not identify deployments that respect QoS expectations and resource constraints.

In SBON [25] they propose an infrastructure that manages and optimizes stream queries from multiple applications. SBON performs an operator placement decision, creating a mapping of operators to physical overlay nodes. This mapping should make efficient use of network resources, for example, by filtering data close to the sources. The SBON uses a decentralized algorithm for network-aware operator placement called Relaxation placement. The idea behind Relaxation placement is to find a solution in two steps. First, an unpinned operator in a query is placed using a spring relaxation technique in a virtual metric latency space. After that, the solution is mapped to actual physical overlay nodes. The function minimized by this approach is the data rate-latency product. This product is the amount of data in transit in the network and thus a measure for global network usage. Moreover, the above solutions do not apply in shared processing environments, since they do evaluate the impact of their decisions on the existing queries of the system.

Finally, Synergy [27] is a middleware for composition of stream-based continuous queries that reuse existing processing components. Although they evaluate the impact of shared components on the QoS of existing queries, their framework addresses the problem of deploying new queries rather that adapting existing deployments to dynamic changes of the network or workload conditions. Thus, they do not support any run-time operator migration.

Resource Allocation. The problem of resource allocation has also been studied in [12] for unstructured overlay-based systems and in [32] for stream processing applications. In [12] they use a fairness index of a distribution as a measure of fairness and load balancing is achieved by replicating documents across multiple nodes in the system. In [32] they present a greedy load distribution algorithm that aims

at avoiding overload and minimizing end-to-end latency by minimizing load variance and maximizing load correlation.

Load shedding techniques [29] have also been proposed in order to reduce processing latency and address shortage in resources, while in [31] they developed algorithms for selecting an operator placement plan that is resilient to changes in load, i.e., plans that will be able to withstand a large set of input rate combinations. Furthermore, in [13] they present a decentralized and adaptiveresourcee allocation algorithm that allows the composition of distributed stream processing applications on the fly, while satisfying their QoS demands. None of the above solution addresses the problem of query deployment in wide-area networks neither employ operator migration and replication to achieve the QoS expectations of the queries.

6. CONCLUSIONS

We introduced an adaptive distributed framework for in-network deployment of shared stream-based queries. The key idea is to identify alternative operator placements which meet the QoS expectations without violating any resource constraints. Our approach allows nodes to react fast to QoS or resource violations by applying one of the precomputed placement configurations. Moreover, metadata regarding the alternative operator deployments can be used to address conflicting concurrent modifications of the shared queries. Our preliminary results shows that our approach is viable.

We are currently optimizing our implementation on PlanetLab, and doing more detailed evaluation of our performance. Moreover, we plan to study further the problem of concurrent modifications and replication-based solutions, e.g., identify the minimum number of replicas required to satisfy all queries. Finally, our plans are based on time-varying statistics, thus we would like to incorporate efficient techniques for updating our metadata, to reflect the most current status of our system.

7. REFERENCES

[1] Distributed intrusion detection, http://www.dshield.org.

[2] Distributed monitoring framework, http://dsd.lbl.gov/dmf.

[3] Earth scope, http://www.earthscope.org.

[4] Yahoo pipes, http://pipes.yahoo.com/pipes/.

[5] Abadi et al. Aurora: A new model and architecture for data stream management. In *VLDB journal*, 2003.

[6] Abadi et al. The design of the Borealis stream processing engine. In *CIDR*, 2005.

[7] Abdelzaher T. An automated profiling subsystem for qos-aware services. In *RTAS*, 2000.

[8] Y. Ahmad and U. Cetintemel. Network-aware query processing for stream-based applications. In *VLDB*, 2004.

[9] Basu et al. Nodewiz: Peer-to-peer resource discovery for grids. In *GP2PC*, 2005.

[10] Campbell et al. IrisNet: an internet-scale architecture for multimedia sensors. In *MM*, 2005.

[11] Chen et al. Gates: A grid-based middleware for processing distributed data streams. In *HPDC*, 2004.

[12] Y. Drougas and V. Kalogeraki. A Fair Resource Allocation Algorithm for Peer-to-Peer Overlays. In *Proceedings of the 8th IEEE Global Internet Symposium*, 2005.

[13] Y. Drougas, T. Repantis, and V. Kalogeraki. Load Balancing Techniques for Distributed Stream Processing Applications in Overlay Environments. In *9th IEEE International Symposium on Object- and Component-Oriented Real-Time Distributed Computing (ISORC 2006)*, 2006.

[14] M. J. Franklin, B. T. Jnsson, and D. Kossmann. Performance Tradeoffs for Client-Server Query Processing. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 25(2):149–160, 1996.

[15] L. M. Haas, D. Kossmann, E. L. Wimmers, and J. Yang. Optimizing Queries across Diverse Data Sources. In *VLDB*, 1997.

[16] Huebsch et al. Querying the internet with PIER. In *VLDB*, 2003.

[17] Jain et al. Design, implementation and evaluation of the linear road benchmark of the stream processing core. In *SIGMOD*, 2006.

[18] D. Kossman. The state of the art in distributed query processing. *ACM Computing Surveys (CSUR)*, 32(4):422–469, December 2000.

[19] V. Kumar, B. F. Cooper, and K. Schwan. Distributed Stream Management using Utility-Driven Self-Adaptive Middleware. In *2nd IEEE International Conference on Autonomic Computing (ICAC)*, 2005.

[20] Kumar et al. Resource-aware distributed stream management using dynamic overlays. In *ICDCS*, 2005.

[21] Kumar et al. IFLOW: Resource-aware overlays for composing and managing distributed information flows. In *EuroSys*, 2006.

[22] Kuntschke et al. StreamGlobe: Processing and sharing data streams in grib-based P2P infrastructures. In *VLDB*, 2005.

[23] L. F. Mackert and G. M. Lohman. R* optimizer validation and performance evaluation for local querie. In *SIGMOD*, 1986.

[24] Motwani et al. Query processing, approximation, and resource management in a stream management system. In *CIDR*, 2003.

[25] Pietzuch et al. Network-aware operator placement for stream-processing systems. In *ICDE*, 2006.

[26] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network . In *SIGCOMM*, 2001.

[27] Repantis et al. Synergy: Sharing-aware component composition for distributed stream processing systems. In *Middleware*, 2006.

[28] Srivastava et al. Operator placement for in-network stream query processing. In *PODS*, 2005.

[29] N. Tatbul and S. Zdonik. Window-aware Load Shedding for Aggregation Queries over Data Streams. In *VLDB*, 2006.

[30] A. Tomasic, L. Raschid, and P. Valduriez. Scaling Heterogeneous Databases and the Design of DISCO. In *ICDCS*, 1996.

[31] Y. Xing, J.-H. Hwang, U. Cetintemel, and S. Zdonik. Providing Resiliency to Load Variations in Distributed Stream Processing. In *VLDB*, 2006.

[32] Y. Xing, J.-H. Hwang, and S. Zdonik. Dynamic Load Distribution in the Borealis Stream Processor. In *ICDE*, 2005.

[33] Yalagandula et al. s^3: A scalable sensing service for monitoring large networked systems. In *INM*, 2006.