Adaptive In-Network Query Deployment 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

In-network processing has been an important research problem for distributed stream processing systems. In this paper, we study the problem in the context of shared processing environments, where query operators could be used by multiple applications. In these environments, run-time modifications on the query deployment add new challenges. Applications with strict and potentially conflicting QoS requirements may share operators. Hence, operator placement decisions must be fast, adaptive to network conditions, and well-coordinated in order to guarantee the QoS expectations.

We propose a novel *sharing-aware* middleware for in-network processing that achieves fast adaptivity to dynamic changes. We follow a proactive approach where nodes propagate metadata regarding *alternative* operator placement configurations. Whenever QoS violations occur, the metadata enables nodes to make fast, localized, operator migration decisions that can adapt to dynamic network conditions and resolve the existing violations.

1. INTRODUCTION

Stream processing systems (SPSs) have gained considerable attention in a wide range of applications including planetary-scale sensor networks [3, 10], network monitoring [1, 2], and feed-based information mash-ups [4]. These applications are characterized by geographically dispersed data sources and consumers, as well as a large number of simultaneous user queries. In order to facilitate these applications, SPSs distribute their processing across multiple nodes, constituting *distributed stream management systems* (DSMSs).

While various DSMSs [6, 12, 13] and solutions for in-network processing [8, 16, 18] have been proposed, these systems are not designed nor optimized for *shared processing environments*, where processing operators are used by multiple applications. In this paper, we are primarily interested in studying this important optimization dimension. Sharing of processing components has been shown to optimize resource utilization by avoiding redundant computations [14, 17]. Moreover, network traffic is reduced as shared data streams are forwarded to a single processing location. Thus, the processing capacity of the system can be significantly improved by reusing computational results.

Existing work on shared environments has focused on composing stream processing queries by discovering and reutilizing existing processing operators [14, 17]. However, the problem of adaptive in-network processing creates new challenges in shared processing environments. First, applications often express Quality-of-Service (QoS) specifications, which describe the utility of the query output (e.g., response delay, end-to-end loss rate). For example, in many real-time financial applications query results are useful only if they arrive timely. However, if sharing is supported, operators will be used by multiple queries with potentially conflicting QoS expectations. Therefore, run-time operator migrations should guarantee the QoS level requested by *every* affected query.

Furthermore, DSMSs are expected to operate over dynamic environments, with large number of unreliable receptors, some or all of which may contribute their resources only on a transient basis (e.g., as in the case of P2P settings). Inherently, in these systems it is difficult to guarantee the QoS requirements of each query. In general, DSMSs should gracefully handle dynamic changes in resource availability and network conditions. Thus, the network deployment of the query set should adapt to dynamic changes and quickly address any QoS violations. Because of the many processing dependencies that are present in shared environments, deployment modifications must be well-coordinated in order to satisfy all QoS expectations.

In this paper, we propose a novel, *adaptive*, and *sharing-aware* middleware that distributes stream-based queries and addresses the aforementioned challenges. In our approach, nodes maintain and propagate metadata regarding *alternative* deployments of the processing operators and periodically apply the best available deployment. Every deployment configuration reflects the current network conditions while it respects the resource constraints of the nodes and the QoS expectations of the queries it affects. A distinguishing feature is that, whenever QoS violations occur, nodes rely on the metadata to make *fast, localized*, and *light-weight* operator placement decisions that can resolve any existing violations, making our solution suitable for QoS-sensitive applications.

2. SYSTEM MODEL

In this section, we present the processing model, the components, and the run-time functionality of our middleware.

Stream processing model. Our middleware streams data from sources to end-clients via in-network processing operators, an example of which is shown in Figure 1. Data sources produce streams, while clients express stream-based continuous queries defined as directed, acyclic graphs (DAGs) of stream-oriented operators [5, 15]. In our system, each source and client has a proxy running on a node, acting on behalf of its corresponding entity. The total set of queries is called the *query network* and also has the structure of a DAG.

We assume that clients can define their queries on any collection of existing streams. This allows for operators and intermediate results to be shared by multiple queries. We fol-

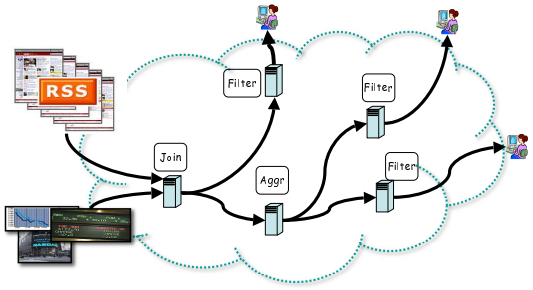

Figure 1: Query deployment example.

low the semantics of a publish-subscribe system in order to ensure that multiple operators can subscribe to another operator's output stream. Hence, operators with multiple inputs have multiple publishers (e.g., union). Moreover, since the output of an operator can be shared by multiple downstream operators, operators may have multiple subscribers.

Each processing operator has specific requirements, i.e., cpu, memory, disk space, etc, which can be obtained through resource profiling techniques [7]. In this work, we focus on cpu requirements, measured in cpu cycles per stream tuple, and we refer to this as the *operator cost*. Clients can also express their *QoS expectations* for the queries they register, such as end-to-end latency, loss rate, etc. We use end-to-end latency as our QoS metric. However, without loss of generality, latency can be replaced with another additive metric.

Operators are free to roam in the network and may be reallocated over time as part of our adaptation process. The placement of a query's operator set defines the *deployment plan* of the query. Operators are interconnected by overlay links, each forwarding the output of an operator to its downstream operators in the query network. Thus, the deployment of the query network creates an overlay topology consistent with the data flow of the query network. In this overlay network, a *downstream* or *upstream neighbor* of a node n, processing an operator o, is a node that processes a downstream or upstream operator of o, respectively. Depending on the available resources and the QoS requirements, there could be multiple alternative deployment plans of the query network.

System architecture. Our system consists of a set of nodes providing stream processing services [5, 15]. We assume that each node has a certain cpu capacity, thus, distributing operators in the network affects the residual capacity of the available nodes. To discover available processing resources, we rely on a distributed *resource discovery service*, implemented and maintained by the nodes in our system.

Our system employs NodeWiz [9], a scalable tree-based overlay infrastructure for resource discovery. Nodes advertise 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 advertisement and query load is balanced across nodes. In our system, nodes periodically advertise their residual capacity to NodeWiz and query the directory to discover available nodes to process the operator set. The resource discovery queries are issued when needed as part of our adaptation process.

Our system relies also on a *network monitoring service* for collecting network statistics of the overlay links between nodes. We use S^3 [19], a scalable sensing service for real-time and configurable monitoring of large networks. The infrastructure measures both network and node metrics, while it uses inference algorithms 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. Since our QoS metric is query response latency, we used S^3 to collect the latency of the overlay links.

Query registration. Clients join the system and register their queries through their proxy which tries to identify if any of the query's operators and streams already exists. Discovery of processing components and data streams was addressed in [14, 17] and it is outside the scope of this paper. In this work, we assume that all available operators are advertised to our resource directory service using a global naming schema and are located by querying this directory.

Initial query deployment. New operators specified by a query are initially placed on the node closest to the publishers of their inputs. This will provide the best placement, with respect to the response latency of the new query. Once all new operators are deployed, our system attempts to discover alternative plans. These plans will respect the cpu capacity of the nodes and meet the QoS of the new query without violating the QoS of the existing ones. Across all discovered plans, we initially apply the one with the minimum bandwidth consumption.

3. ADAPTIVE OPTIMIZATION

Our system periodically identifies a set of alternative deployment plans that reflect the latest network conditions. It also continuously monitors the current query deployment and addresses any resource or QoS violations through these alternative deployments. In this section we describe our approach.

Goals. The goal of our system is to efficiently *adapt* the deployment of the query set to the network conditions and the resource availability. Any QoS or resource violations should be detected and addressed as soon as possible through fast and light-weight decisions. Finally, since operators are shared by multiple queries, nodes should evaluate the impact of any operator placement decisions on every affected query.

Our approach. To address these goals, we designed a decentralized, *sharing-aware* operator placement protocol that respects the resource constraints of the nodes and the QoS expectations of every query. Our approach relies on localized information and has low computational cost. Each node maintains metadata regarding alternative placements of its *local* operators and periodically propagates it downstream. Downstream nodes aggregate the metadata to compose and locally store candidate deployment plans that place their local and upstream operators.

At run-time, nodes monitor the network links with their overlay neighbors and identify changes that violate any query's QoS. They adapt to these changes by validating their local deployment plans and applying, through operator migration or replication, a *feasible* (i.e., with no violations) deployment. Nodes break ties across multiple feasible deployments by applying the one with the minimum bandwidth usage. Our approach allows for fast reaction to dynamic changes as it reduces the communicational and computational overhead of identifying from "scratch" and during run-time a valid deployment. Moreover, the system periodically improves the current deployment plan by applying a newly-discovered feasible deployment that reduces the bandwidth usage. Next we provide the definition of a feasible deployment plan.

DEFINITION 1. *Let us assume a set of nodes $\mathcal{N}$ with cpu capacities $z_i, \forall n_i \in \mathcal{N}$ and a set of operators $\mathcal{O}$ with process-*

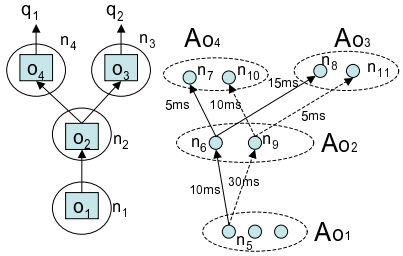

Figure 2: Query example.

ing cost c_j and input rate r_j^{in}, $\forall o_j \in \mathcal{O}$. Given a query set $\mathcal{Q}$ sharing the operator set $\mathcal{O}$, and query QoS expectations $w_t, \forall q_t \in \mathcal{Q}$, a feasible deployment plan *of the operator set $\mathcal{O}$ across the node set $\mathcal{N}$, is one that satisfies the following:*

$$\sum_{o_j \in O_i} c_j \times r_j^{in} \leq z_i, \forall n_i \in \mathcal{N}$$

$$d_t \leq w_t, \forall q_t \in \mathcal{Q}$$

where O_i is the operator set placed on node n_i. The QoS level of a query q_t is the end-to-end response latency of the query, d_t, measured by aggregating the network latency of all overlay links connecting each operator of the query to the publisher of its input. If an operator has multiple publishers, then we use the maximum latency among them.

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. We note as $d(n_i, n_j)$ the network latency of the link between n_i and n_j.

Our solution discovers a set of feasible deployment plans to be used under different QoS violation scenarios. Moreover, we periodically apply the plan that minimizes the bandwidth consumption, i.e., the sum of outgoing data rate of each node. The bandwidth usage of a deployment plan b^p is defined as:

$$b^p = \sum_{o_j \in \mathcal{O}} \sum_{o_m \in sub_j} r_j^{in} \times s_j \times \phi(h^p(o_j), h^p(o_m))$$

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

where the function $h^p(o_j) : \mathcal{O} \rightarrow \mathcal{N}$ provides the placement node of the operator o_j based on the plan p, s_j is the selectivity of the operator o_j, and sub_j is the set of operators that subscribe to o_j's output stream.

3.1 Deployment plan discovery

Our protocol is initiated by the nodes hosting *leaf* operators, i.e., having no upstream operators, and discovers feasible deployment plans incrementally. Each node identifies possible placements of its local operators and evaluates the impact of each placement on the query latency and the bandwidth usage. This metadata is propagated downstream. Downstream nodes aggregate the metadata, generate plans that deploy their local *and* upstream operators, and propagate them downstream. We refer to these as *partial deployment plans.*

Figure 2 shows an example deployment of two queries, q_1 and q_2, that share two operators, o_1 and o_2. The plan generation process is initiated by n_1, which forwards possible placements of o_1 to n_2. Node n_2 will combine these placements with the candidate placements of o_2 in order to generate partial plans that deploy both o_1 and o_2. These plans are forwarded further downstream to nodes n_3 and n_4.

Symbol	Definition
c_i	cost of operator o_i
r_i^{in}	input rate of operator o_i
s_i	selectivity of operator o_i
w_t	QoS of query q_t
d_t	response latency of query q_t
sub_i	subscribers (downstream operators) of o_i
pub_i	publishers (upstream operators) of o_i
$h(o_i)$	host node of operator o_i
z_i	capacity of node n_i
O_i	set of operators hosted on n_i
Q_i	set of queries sharing operator o_i
A_i	candidate hosts of operator o_i

Table 1: Notation.

A unique challenge of our approach is to identify and eliminate, as early as possible in the plan propagation process, partial plans that cannot meet the QoS of the queries they affect, i.e., the queries sharing the operators in the plan. To identify infeasible partial deployments we propose the *k-ahead search* algorithm.

3.1.1 *k-ahead search*

Every node n_v runs the search algorithm for each local operator and each candidate host for that operator. If A_i is the set of candidate hosts of the local operator $o_i \in O_v$, 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 $n_j \in A_i$. Intuitively, if Q_i is the set of queries using operator o_i, the k-search attempts to identify the minimum impact on the latency of each query $q_t \in Q_i$, if we migrate o_i to node n_j and make the best placement decision (w.r.t. latency) for the next k downstream operators of each query q_t. Below we describe 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 $o_i \in O_v, n_j \in A_i, q_t \in Q_i$.

Node n_v executes the following for each operator $o_i \in O_v$:

1. It identifies the candidate hosts A_i of the local operator o_i by querying the resource directory service. Assuming the constraint requirements of o_i are $Z = [(z_1, v_1), (z_2, v_2), ..., (z_m, v_m)]$, where z_i is the resource attribute and v_i is the operator's requirement for that resource, we query the resource directory for nodes with
$$z_1 \geq v_1 \wedge z_2 \geq v_2 \wedge ... \wedge z_m \geq v_m.$$
2. If o_m is the downstream operator of o_i for the query $q_t \in Q_i$, node n_v probes the host of o_m for the set of candidate hosts of o_m, A_m. For each candidate, it queries the networking monitoring service for the latency $d(n_j, n_t), \forall n_j \in A_i, \forall n_t \in A_m$. The 1-ahead latency for the o_i operator with respect to its candidate n_j and the query $q_t \in Q_i$ is:
$$\gamma_i^1(n_j, q_t) = \min_{n_t \in A_m} \{d(n_j, n_t)\}$$
In Figure 2, n_1 requests from n_2 the candidate hosts of o_2, A_2, and will estimate the 1-ahead latencies $\gamma_1^1(n_5, q_1) = \gamma_1^1(n_5, q_2) = 10ms$. Let us also assume for o_2 that $\gamma_2^1(n_6, q_1) = 5ms$ and $\gamma_2^1(n_6, q_2) = 15ms$.
3. The algorithm continues in rounds; for each operator o_i, n_v waits for its subscriber o_m in the query $q_t \in Q_i$ to complete the evaluation of the $(k-1)$-ahead latency. The k-ahead latency for o_i with respect to its candidate

host n_j and the query $q_t \in Q_i$ is:

$$\gamma_i^k(n_j, q_t) = \min_{n_t \in A_m} \{\gamma_i^{k-1}(n_t, q_t) + d(n_j, n_t)\}$$

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, if we migrate o_1 to n_5 and apply the best possible (i.e., with the minimum latency) placement decision of the next two operators, the response latency of q_1 will be increased by 15ms and the latency of q_2 by 25ms.

The k-search algorithm is executed independently of the plan generation process. Moreover, assuming that network latencies change less frequently than stream rates, the algorithm can be executed at a low frequency and its results can be reused by the plan generation process. The plan generation algorithm should run with a frequency that reflects the workload changes. In the next paragraph, we describe how we use the results of the k-ahead search.

3.1.2 Deployment plan generation

Let us assume an operator o_i shared by a set of queries $q_t \in Q_i$ and let pub_i be the set of upstream operators for o_i. For the example of Figure 2 $pub_3 = pub_4 = \{o_1, o_2\}$. A partial deployment plan for o_i assigns each operator $o_j \in pub_i \cup \{o_i\}$ to a network node. Each partial plan p is associated with metadata, namely (a) a *partial cost*, pc^p, i.e., the bandwidth consumption it occurs, and (b) a *partial response latency* for each query it affects, $pl_t^p, \forall q_t \in Q_i$. For example, a partial plan for o_2 will assign o_1 and o_2 to two nodes and evaluate the bandwidth usage of these placements and the output latency up to operator o_2. In the rest of the section, we describe how partial deployment plans are created by the leaf nodes and we then focus on the nodes executing intermediate operators.

Leaf nodes. Let o_i be a leaf operator processed on n_v. Node n_v obtains the candidate hosts of o_i from the resource discovery service, generates a set of partial plans, and evaluates their partial cost and the partial latencies of all queries sharing o_i. Each plan assigns o_i to a different candidate n_j. Since this plan assigns only the first operator, its partial cost is zero. If S_{o_i} is the set of input sources for o_i and $h(s)$ is the node publishing data on behalf of the source $s \in S_{o_j}$, then the partial latency (i.e., the latency from the sources to n_j) of a query q_t using operator o_i is:

$$pl_t^p = \max_{s \in S_{o_i}} d(h(s), n_j), \forall q_t \in Q_i.$$

Plan elimination. We eliminate any partial plans that could not lead to a feasible deployment and propagate downstream the remaining ones. 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 output latency of the 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. 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,

$$pl_t^p + \gamma_i^k(n_j, q_t) \geq w_t. \quad (1)$$

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. Hence, there is a tradeoff with respect to the parameter k. The more operators ahead we search, the earlier we will discover infeasible plans; however, the k-ahead search will incur higher overhead. Our initial experiments reveal that our approach manages to eliminate a significant number of redundant partial plans early in the propagation process.

Intermediate nodes. Non-eliminated partial plans are propagated downstream along with their metadata (i.e., partial cost and latencies). Let us assume that node n_v, processing o_i, receives a partial plan p from the host of its upstream operator o_m. For purposes of illustration we assume a single upstream publisher. Our equations can be generalized for many publishers in a straightforward way.

Queries sharing o_i also share its upstream operators. Thus, each received partial plan p already includes a partial latency $pl_t^p, \forall q_t \in Q_i$. Node n_v expands these plans by adding candidate placements for o_i. For example, in Figure 2, n_2 will receive partial plans that deploy o_1 and expand them with possible placements for o_2, before propagating them to n_3 and n_4. Received partial plans include their partial cost and latency for q_1 and q_2 based on each plan's configuration.

Node n_v first validates the residual capacity of each candidate host of the operator o_i, $n_j \in A_i$: it parses each received partial plan p and checks if any upstream operators have also been assigned to n_j. To facilitate this, we send along with each plan the expected load requirements of each operator included in the partial plan. If the residual capacity of n_j is enough to process all assigned operators, we create a new partial plan that expands the plan p with the assignment of o_i to n_j. The partial metrics of this new partial plan f are:

$$pl_t^f = pl_t^p + d(h^p(o_m), n_j), \forall q_t \in Q_i$$
$$pc^f = pc^m + r_m^{out} \times \phi(h^p(o_m), n_j)$$

The partial plan f is propagated if the k-ahead latency $\gamma_i^k(n_j, q_t)$ can satisfy the Equation 1.

Final plan generation. Partial plans created on a node are "finalized" and stored locally. A partial plan is finalized by quantifying its impact on the current bandwidth usage and on the QoS of the queries it affects. To facilitate this process nodes collect statistics on the bandwidth consumed by their upstream operators and the query latency up to their local operator. For example, in Figure 2, n_2 estimates the bandwidth consumption from o_1 to o_2 and the latency up to operator o_2. For each partial 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 QoS of the queries. In Figure 2, n_2 stores plans that place operators $\{o_1, o_2\}$, while n_4's plans deploy the set $\{o_1, o_2, o_4\}$.

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 QoS we keep the ones with the minimum bandwidth consumption, while if they have similar impact on the bandwidth consumption we store the ones that improve the QoS the most. We omit the details due to space limitations.

3.2 Handling QoS violations

Our system periodically initiates the plan generation process and discovers deployments that reflect the current network conditions. If we discover alternative deployments that can improve the bandwidth consumption, we apply the one with the best impact. Moreover, nodes store their local precomputed deployments plans and use them whenever changes in their "close neighborhood" violate the QoS requirements.

Detecting violations. Nodes detect QoS violations by monitoring the latency to their upstream operators, i.e., their publishers. Moreover, they periodically receive the output latency of all queries sharing their local operators and they quantify their "slack" from their QoS expectations, i.e., the latency increase each query can tolerate. Let us assume an operator o_i used by a query q_t with slack $slack_t$. If the latency of the link between o_i and its publisher o_m increases by:

$$\Delta d(h(o_m), h(o_i)) > slack_t,$$

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

Identifying recovery plans. To resolve this violation, the host of o_i searches all local deployment plans for a plan p that migrates both o_i or o_m (i.e., removes the bottleneck link) and decreases enough the current latency of q_t, d_t:

$$w_t - d_t \leq \Delta pl_t^p \leq 0$$
$$\Delta pl_t^p + \Delta d(h(o_m), h(o_i)) \leq w_t - d_t$$

Across these plans, we apply the one with that has the best impact on the bandwidth consumption.

Occasionally, a deployment plan satisfying q_t can not be locally discovered. In this case, the host of o_i sends a request for a "recovery" plan to its downstream operator in the violated query q_t, along with information for the congested link (e.g., its new latency). Nodes that receive such requests, search in their local plans (in the same way described above) for an alternative deployment that can satisfy the QoS of 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.

Recovery overhead. 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 by their impact on each query's latency. Thus, when a QoS violation occurs, nodes can identify a "recovery" deployment plan very fast.

3.2.1 Concurrent QoS violations

Often, simultaneous QoS violations could be detected in parallel by different nodes, which will attempt to apply their own "recovery" deployment plan. This could lead to conflicting operator placements with undefined impact on the affected queries. For example, in Figure 2, if the latency requirements of q_1 and q_2 are not met, nodes n_3 and n_4 may decide simultaneously to apply a different deployment plan.

Nodes rely on the metadata created by the plan generation process to identify and resolve these cases. Specifically, the metadata enables nodes to (i) decide whether simultaneous requests for different deployment plans are creating any conflicts, (ii) build non-conflicting deployment plans that can guarantee the QoS of multiple queries. Furthermore, if a non-conflicting plan cannot be discovered, we apply the "recovery" plans by *replicating* the operators to the new location.

Network size	20	40	60	80	100
Random	49.16%	44.25%	40.33%	45.14%	45.66%
Sharing-aware	81.82%	91.24%	97.65%	100%	100%

Table 2: Percentage of satisfied QoS expectations.

To detect concurrent modifications of the current operator deployment, we follow a lock-based approach. Once a node decides that a new deployment should be applied, all operators in the plan and their upstream operators are locked and informed about the plan to be applied. Nodes trying to migrate already locked operators need to check if their operations does not conflict with the current one in progress. If a conflict exists, they attempt to identify an alternative non-conflicting deployment. Otherwise, they apply their initial plan by replicating the operators. Due to space limitations, we briefly discuss how we identify conflicting deployment plans.

Non-conflicting plans. Concurrent modifications are not always conflicting. If two deployment plans decided by different nodes do not affect 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, then both changes can be applied. In this case, the two plans decided by n_3 and n_4 will show no impact on the queries q_1 and q_2 respectively. The metadata stored with the deployment plans includes the information (operators to be migrated, new hosts, impact on the queries) to identify cases where multiple non-conflicting plans can be applied in parallel.

Moreover, multiple deployment plans may not be necessary in order to resolve simultaneous QoS violations. Often, a *single* plan might be sufficient in order to reconfigure the current deployment. In our approach, every plan includes the impact on all affected queries. Thus, if two plans are affecting all violated queries, then either one can provide a feasible deployment. Therefore, our system will apply the plan that first acquires the migration lock and will ignore subsequent requests that affect the violated queries.

Resolving conflicting plans. Deployment plans that relocate shared operators cannot 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, we search at the hosts of its downstream operators to discover plans that include 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.

4. PRELIMINARY EVALUATION

We developed an initial prototype in Java and run 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 [19]. All nodes are organized in the NodeWiz resource discovery overlay and 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.

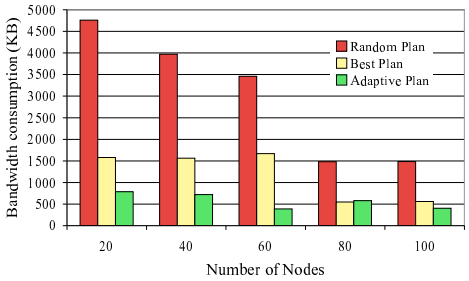

Figure 3: Total bandwidth consumption.

To create shared queries, we treat each operator as a separate query; if an operator is in the i^{th} 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. RSS sources are assigned to a random set of nodes which publish their feeds. Clients are also hosted by random nodes.

Table 2 shows the percentage of queries that meet their QoS request for two different deployment approaches and different network sizes. *Random* assigns each operator to a random node with sufficient cpu capacity. *Sharing-aware* uses our framework to discover feasible deployments and applies 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 are available, we are able to satisfy all queries. We also manage to decrease the average query latency, compared with the Random deployment, by 45%-52%, depending on the network size and workload. We omit the figures due to space limitations.

To demonstrate our system's adaptivity we used the followed approach. We acquire a set of feasible deployment plans and for each operator chain we initially apply a random *feasible* plan (*Random Plan*). Note, that this is a different deployment than the previous Random, as it takes also QoS expectations into consideration. Once all operators are deployed, our system identifies better feasible plans and gradually changes the deployment to meet these best plans, achieving the bandwidth consumption shown by *Best Plan* in Figure 3.

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 3 shows the results. *Best plan* achieves a significant improvement over *Random Plan*. Moreover, when we decrease the input rate, the workload of the nodes decreases. Thus, we discover better deployments, which assign sequences of operators to the same node, reducing in-network stream forwarding. Finally, as the network size increases more resources become available, thus, larger operator chains are processed on a single node, decreasing further the bandwidth usage.

5. RELATED WORK

Recently, there have been a number of SPSs [5, 12, 15] and frameworks for in-network deployment of continuous queries [6, 11, 13]. Borealis [6] supports shared processing and operator migration, while GATES [11] avoids run-time operator migrations by assigning operators to predefined locations. None of these approaches attempts to identify deployments that respect QoS expectations and resource constraints.

Moreover, the operator placement problem has been studied in [8, 13, 16, 18]. SAND [8] and SBON [16] propose network-aware algorithms that improve the bandwidth usage. In [13] they propose a DSMS that distributes processing based on network data aggregation, while in [18] they assume a hierarchical stream collection which is not applied in our architecture. These solutions cannot be applied in shared processing environments, since they do not evaluate the impact of their operator placement on the QoS of existing queries.

Finally, Synergy [17] is a middleware for composition of stream-based queries that reuse existing processing components. Although they evaluate the impact of sharing components on the QoS of existing queries, they address the problem of deploying new queries rather that adapting existing deployments to dynamic changes of the network conditions. Thus, they do not support any run-time operator migration.

6. CONCLUSIONS

We introduced an adaptive middleware for in-network deployment of shared stream-based queries. The key idea is to proactively identify alternative operator placements and address run-time QoS or resource violations by applying the best deployment plan. In terms of future work, we have a full agenda. We are currently finalizing our implementation and we plan to deploy it on PlanetLab. This will allow us to have a detailed system evaluation. Moreover, we plan to further explore the problem of conflicting QoS requirements and replication-based solutions, e.g., identify the minimum number of replicas required to satisfy all queries. plan to incorporate techniques for incrementally updating the alternative plans.

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] Yanif Ahmad and Ugur 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] Jain et al. Design, implementation and evaluation of the linear road benchmark of the stream processing core. In *SIGMOD*, 2006.
[13] Kumar et al. Resource-aware distributed stream management using dynamic overlays. In *ICDCS*, 2005.
[14] Kuntschke et al. StreamGlobe: Processing and sharing data streams in grib-based P2P infrastructures. In *VLDB*, 2005.
[15] Motwani et al. Query processing, approximation, and resource management in a stream management system. In *CIDR*, 2003.
[16] Pietzuch et al. Network-aware operator placement for stream-processing systems. In *ICDE*, 2006.
[17] Repantis et al. Synergy: Sharing-aware component composition for distributed stream processing systems. In *Middleware*, 2006.
[18] Srivastava et al. Operator placement for in-network stream query processing. In *PODS*, 2005.
[19] Yalagandula et al. s^3: A scalable sensing service for monitoring large networked systems. In *INM*, 2006.