XFlow: Internet-scale Distributed Stream Processing

Olga Papaemmanouil, Ugur Cetintemel and John Jannotti

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-07-06
June 2007

Integrating Pub-Sub and Stream Processing for Internet-Scale Monitoring

Olga Papaemmanouil, Uğur Çetintemel, John Jannotti
Department of Computer Science, Brown University
{olga, ugur, jj}@cs.brown.edu

ABSTRACT

Existing stream processing systems are designed and optimized for clustered deployments, and cannot adequately meet the scalability and adaptivity requirements of Internet-scale monitoring applications. Furthermore, these systems commonly optimize for a specific QoS metric, which may limit their applicability to diverse applications and environments.

This paper presents the high-level design, architecture, and evaluation of XFlow, a generic distributed data collection, processing, and dissemination system that addresses these limitations. XFlow integrates an extensible pub-sub model with data flows for stream processing. The underlying pub-sub model decouples stream sources and clients, as well as the processing operators, leading to a loosely-coupled architecture that can gracefully scale, adapt to churn in system membership and workload, and facilitate sophisticated optimizations.

We first provide an overview of the basic design and architecture of XFlow. We then describe XFlow's extensible optimization model that changes the placement and implementation of operators to meet application-specific performance goals and constraints. Finally, we demonstrate the flexibility and the effectiveness of XFlow using real-world streams and experimental results obtained from up to 150 PlanetLab sites.

1. INTRODUCTION

Motivation and goals: The confluence of ubiquitous, high-performance networking and increased availability of hardware/software receptors that report physical or software events has led to the emergence of a new class of highly-distributed, large-scale applications, which we collectively refer to as Internet-Scale Monitoring (ISM). An ISM application is a networked system that consists of large numbers of geographically dispersed entities: sources that generate potentially large volumes of data streams and consumers that register large numbers of concurrent queries over these data streams. Example ISM applications include planetary-scale sensor networks or "macroscopes" [3, 10], network performance and security monitoring [1, 2], massively multi-player online games, and feed-based information mash-ups [4]. Going forward, there is a need for general-purpose ISM "infrastructures" that can effectively support a broad spectrum of ISM applications. We introduce such an ISM infrastructure, named XFlow, which is an extensible, data stream acquisition, processing and distribution system.

Several high-level goals and capabilities guided the design of XFlow:

- *Scalability*: ISM 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 numbers of simultaneous user queries. To achieve both types of scalability, the system should scale out and distribute its processing across multiple nodes.
- *Adaptivity*: ISM systems are expected to operate over the public Internet, with large numbers 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, the system should be capable of dealing with churn, time-varying workload, and resource availability.
- *Extensibility*: While many ISM applications share common characteristics (*e.g.*, stream-based processing, overlay networks, membership management), they often exhibit diverse application-specific logic and performance requirements and constraints. For example, a camera-based surveillance application may need to perform feature extraction over MPEG streams, whereas a feed-oriented application may involve XPATH queries over RSS streams. Similarly, a network intrusion application may have strict result latency requirements, whereas an environmental monitoring application running in a peer-to-peer setting may care more about fairness in bandwidth consumption. An ISM system should be easily customized to support application-specific data types, processing logic, performance expectations, and constraints.

XFlow: Integrating pub-sub and stream processing: Distributed pub-sub systems [12, 13] address some of the above requirements. These systems effectively decouple data sources and destinations over geography and time: sources publish data without knowing where and when the consumers will access them and consumers declaratively express their profiles and receive matching data without knowledge of specific data sources. This loosely-coupled architecture allows for high scalability and robustness in the presence of churn. Moreover, extensible pub-sub models have been proposed [23], allowing applications to define their own performance expectations. However, pub-sub systems have traditionally only supported *stateless* queries (*e.g.*, predicate-based filters over individual events) and simple data transformations.

Existing stream processing systems [5, 21], on the other hand, support complex *stateful* continuous queries. Distributed versions [6, 8, 24] allow processing to be transparently distributed across multiple nodes to improve system scalability and availability. However, most current implementations implicitly assume a small number of data sources and

destinations, and, thus, do not provide mechanisms for "massively" distributing processing to take advantage of the large number of computing elements available in the network (e.g., via operator replication and partitioning). Furthermore, these systems often have one or a few QoS metrics of interest, which they optimize using hard-wired approaches. Implementing mechanisms for new metrics is a challenge when the built-in metric is not the "right one" for a given application.

Our experience with both pub-sub (XPORT [23]) and stream processing systems (Aurora/Borealis [5, 6]) convinced us that a hybrid approach has the potential to yield the best of both worlds. For example, while addressing the limitations of our initial Borealis implementation that tightly coupled operators and nodes, we found ourselves implementing a pub-sub-like mechanism to enable easy addition and removal of queries and dynamic operator migration. We also found it extremely difficult to effectively incorporate new optimization metrics and algorithms: each new addition took almost as much as the last one to realize and test.

This paper describes the outcome of our efforts to seamlessly integrate an extensible, distributed pub-sub model with a stream-oriented continuous query model, a combination we argue has the potential to meet the aforementioned design goals. In XFlow, sources publish their data and clients subscribe their processing needs through continuous queries consisting of stream-oriented operators [5, 21]. It is the responsibility of the system to collect and process the data and distribute the results to the clients, while meeting application-specific performance expectations and constraints. This unique combination leads to a loosely-coupled, flexible stream-processing architecture that can scale, adapt and be easily customized for specific target applications.

From an operational perspective, XFlow creates, maintains and optimizes a pub-sub overlay network, given dynamic stream sources, clients with stream-oriented complex profiles, application-specific performance expectations, and available broker machines. The broker network consists of multiple, potentially overlapping overlay trees, in principle one for each source. XFlow uses a generic cost model, capable of expressing a variety of useful cost functions. At run-time, XFlow iteratively modifies the placement and execution of existing query operators to reduce the system cost and meet constraints. The optimizations are guided by a set of operator distribution operations, which include migration, replication and partition. XFlow employs low-overhead mechanisms to evaluate the benefits of these operations.

One of the key features of XFlow is that it uses the underlying pub-sub model to also decouple the query operators. XFlow treats operators as regular stream sources and consumers—each operator subscribes to the stream generated by its upstream operator in the data flow and also publishes the stream it produces. This approach unifies and simplifies the overall system model while at the same time facilitates (i) the sharing of intermediate processing results and (ii) dynamic query modifications such as adding, removing, migrating, and replicating operators.

Contributions and outline: We built XFlow on top of the distributed, extensible pub-sub functionality offered by XPORT and the core stream processing model of Borealis. This unique integration resulted in the following novel features that neither XPORT nor Borealis (or any other system that we are aware of) possesses:

1. A system model, consisting of multiple, potentially overlapping broker overlay trees, for data collection, processing, and result dissemination (note that previous work showed the benefits of the multi-tree approach for various applications [18, 22]).
2. A generic costing model that can express a range of cost metrics and constraints which evaluate the overhead and efficiency of user queries.
3. A cost-metric-agnostic distributed optimization protocol that modifies *where* and *how* the query operators are executed to optimize system behavior.

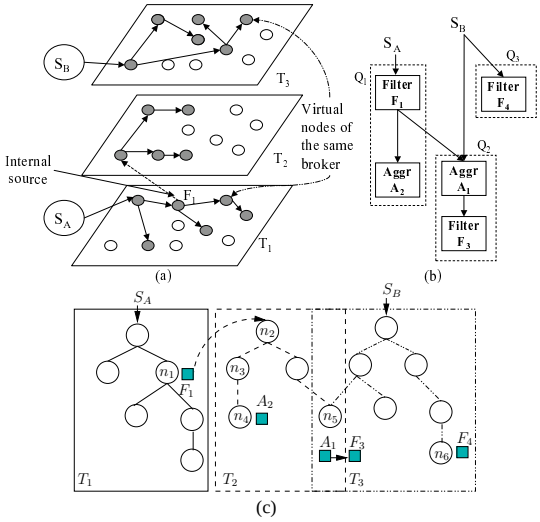


Figure 1: XFlow's system model.

We demonstrate the effectiveness, flexibility, practicality of XFlow through prototype-based experimental results obtained by using real-world web feeds and deployment on up to 150 PlanetLab sites.

The rest of the paper is structured as follows. We describe the basic system architecture of XFlow in Section 2. We introduce how performance metrics can be defined in Section 3 and describe our generic optimization framework in Section 4. We present our experiments and results in Section 5, describe related work in Section 6, and conclude the paper with final remarks and plans for future work in Section 7.

2. SYSTEM MODEL

XFlow consists of an overlay network of a large number of cooperating *brokers* (or nodes), providing stream routing and stream-based query processing services (*e.g.*, [5, 21]). *External data sources* reside outside system boundaries and publish data streams according to a well-defined global schema. Clients (or users) are also external and are eventual consumers of query results: they subscribe their data interests expressed in terms of stream-oriented continuous queries on the global schema. Each external source and client has a proxy running on a broker, acting on behalf of its corresponding entity. In the rest of the discussion, we use node and broker, interchangeably.

Publish-Subscribe model. XFlow relies on the pub-sub model as the underlying uniform mechanism for disseminating *all* data flows in the system. One implication is that each active query operator publishes its output stream, while subscribing to its input stream(s). As operators also publish data, we refer to them as *internal data sources*. Both external

and internal sources are assigned system-wide unique identifiers and have well-defined schemas. The global schema is the union of external and internal schemas.

XFlow creates multiple overlay dissemination trees, potentially one for each internal or external source. Each source publishes its stream to an overlay tree which distributes it to the subscribed consumers. Hence, nodes hosting interested consumers (clients or operators) must join the tree. The main advantage of using multiple trees is to enable better network utilization and reduce redundant transmissions by having clients become part of only those trees that publish relevant data. Figure 1 (a) shows a network of three overlapping trees. Trees T_1 and T_3 publish the external sources' streams S_A and S_B respectively, while T_2 publishes the output of the internal source F_1.

Source Registration. When a new source (internal or external) is first registered to the system, it forwards the schema of its stream to the registration service (referred to as *bootstrap node*). For an external source, the bootstrap is contacted through its proxy. The bootstrap picks a broker as the *root broker* for that data stream. The choice of the root broker should be based on its topological distance to the data source, the available bandwidth of the broker and the expected data volume. This broker will act as the root of the tree that disseminates the source's data stream. The bootstrap's functionality can be easily distributed across multiple nodes to improve the availability and scalability of the system.

Query registration. To subscribe, the client's host contacts the bootstrap and requests a list of the streams published by both external and internal sources. Users can inspect these schemas and define their queries on any collection of external or internal sources. This allows for intermediate results to be shared by multiple queries. For example, in Figure 1(b) query Q_2 uses the output of operator F_1 of query Q_1. Clients are responsible for identifying shared interests with existing queries; *i.e.,* as with Borealis, or even Yahoo pipes [4], XFlow currently requires the user to browse the existing query network to manually identify common sub-queries, although it can also incorporate techniques for automatic detection of sharable computations [19].

The bootstrap is responsible for informing the client's host about the trees publishing its query's inputs. The host joins these trees, creating a new *instance* for each tree, referred to as a *virtual node*. The virtual node connects randomly to one node in the tree and forwards a request for the query's input streams published by that tree. We refer to this request as the *profile* of that virtual node and is extracted from the selection predicates of the query. This profile is forwarded upstream to the root, creating a reverse routing path. Using the routing tree created, a virtual node can route streams published through this tree only to its children interested in receiving them.

Stream processing model. User profiles are expressed as directed, acyclic data-flow graphs of stream-oriented operators (*e.g.*, [5]), operating over the global schema. XFlow has a built-in set of standard windowed operators (filters, unions, aggregates, joins) and also allows for arbitrary user-defined functions to be linked as operators. An example of three simple queries is shown in Figure 1(b).

When an external client subscribes by forwarding its query to a node, the operators of the query get subscribed to their input streams. This is done along the upstream to downstream direction, in an order that is consistent with a topological sort of the query operators in the query plan.

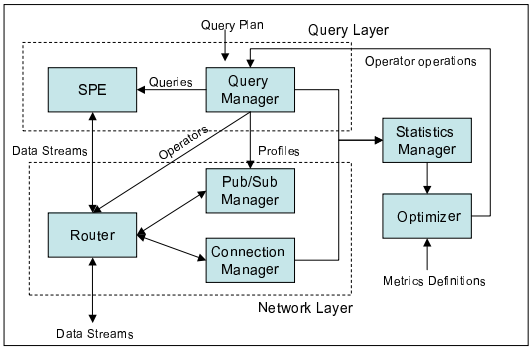


Figure 2: XFlow node architecture.

While the external source and client proxies are "pinned" to the brokers with which they first registered, the query operators are free to roam. All operators of a given query are initially assigned to the same broker; however, as we describe below, operators may be relocated, partitioned or replicated over time as part of the optimization process.

Figure 1(b) and (c) shows three queries and a possible distribution of their operators across the network. Q_1's operators are distributed across two trees. F_1's host, n_1, joins tree T_1, to receive stream S_A, and publishes its output to tree T_2. Hence, n_4, the host of A_2, joins T_2. Node n_5, the host of Q_2, joins also T_2. Moreover, it joins T_3 and subscribes to stream S_B. Finally n_6, hosting F_4, joins tree T_3.

Node architecture

The architecture of an XFlow node (Figure 2) reveals how we integrated the pub-sub model with a stream processing system (SPE). Incoming query plans are parsed by the *query manager* and queries are sent to the SPE for execution. Profiles are extracted and pushed to the *pub/sub manager* which contacts the bootstrap and requests the set of trees publishing the streams specified by the profile. The node joins the tree through the *connection manager* that maintains the connections with the node's parents and children. The *router* component delivers input streams to the SPE and output streams to clients or downstream nodes in the trees. It also routes the maintenance traffic required by the connection manager as well as any profile updates sent by the pub/sub manager.

The *optimizer* is the extensible component in our architecture. Application designers can specify the own performance criterion (see details in Section 3), and XFlow customizes its functionality based on these metrics. Using statistics on the query and the overlay network, obtained from the *statistics manager*, it identifies optimization operations on the queries that improve the cost metrics. Changes on the queries are sent to the query manager and any operators to be reallocated are migrated to their new locations.

Tree management

In principle, XFlow creates one dissemination tree per each source (internal or external). In practice, we employ simple techniques to control the number of trees. First, we allow the definition of super-operators that combine multiple connected operators in a single unit. In such a case, the constituent operators are transparent and will not publish data. Thus, only a single tree will be created for each super-operator.

Second, sources can be assigned to existing trees, reducing the number of dissemination trees. For example, sources re-

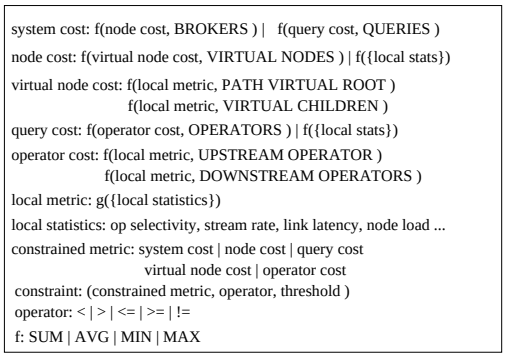


Figure 3: Cost metric grammar.

quested by highly overlapping sets of subscribers can publish through the same tree, while streams can be periodically reassigned across root brokers, adapting to membership changes. The problem of effectively grouping data sources and mapping them to a given set of trees has been studied in [7, 22] and is beyond the scope of this paper. For simplicity of exposition, we assume that this grouping is already done and use source to mean collections of sources.

In order to further reduce the overhead of maintaining multiple trees, one network-level optimization we perform is to have any two pairs of brokers communicate through only a single TCP connection, independently of the number trees in which they are neighbors. These connections create an overlay mesh on top of which "logical" dissemination trees are built, that share the overlay links of this mesh. Hence, the cost of creating a new tree is quite small, since nodes will need to set up connections with their peers only the first time they need to connect on some tree.

3. EXTENSIBLE COST-MODEL

The cost-model of XFlow was derived from that of XPORT, which provides a grammar that allows application designers to specify performance metrics and constraints using a two-level aggregation model: the first layer defines a cost metric for each node and the second layer aggregates this metric over all nodes (of a single overlay tree) to calculate the global system cost. We refer to these metrics as *node-oriented* as they evaluate the overhead the dissemination process incurs on each node.

XFlow builds on and extends XPORT's basic cost model along two non-trivial dimensions. First, it incorporates high-level *query-oriented* metrics that can express the overhead and efficiency of user queries. Second, it allows both query-oriented and node-oriented metrics to be defined and evaluated across a broker network of *multiple* overlay trees, whereas XPORT limits itself to a single tree.

In the rest of the section, we describe this extended grammar in detail (Figure 3) and provide examples.

3.1 Cost metric model

We start with a brief description of XPORT's approach [23] and then continue with the new extensions.

XPORT nodes maintain a set of built-in *local metrics* (e.g., link latency, incoming data rate, etc.), while applications can also provide their own methods for calculating other local metrics. Each node is configured to collect the local metrics of its children or the nodes on its path to the root in the tree. A node's cost is defined as an aggregation of these metrics and the costs of all nodes are aggregated to define the system cost.

In XFlow, the built-in local metrics we provide include statistics on local operators, *e.g.*, input/output stream rates, selectivity, processing costs, etc. Application designers can also combine these statistics and define their own local metrics for the nodes, *e.g.*, the processing load of a node.

Node-oriented metrics. For node-oriented metrics, we introduced one more aggregation level to allow metric definitions for multiple trees. In XFlow, the first level defines the cost of a *virtual* node, *i.e.*, its cost in a single tree, identical to the way a node cost is defined in XPORT: we aggregate local metrics from the virtual node's children or path to the root of the tree. The second level defines the cost of a node, *i.e.*, its cost across all trees, by aggregating the cost of all its virtual nodes. The global system performance is then defined in the new third level by aggregating the cost of all nodes.

An example metric is the average outgoing bandwidth across nodes. The cost of a virtual node is the total data it forwards to its children in the tree. The node cost is then computed as the sum of the data each virtual node sends, plus the data sent to its clients or published to another tree, *i.e.*, the output rate of its local operators. We define an operator's output rate as a local metric, given by the product $r \times s \times k$, where r is the incoming stream rate (tuples in time unit), s the selectivity of the operator and k the average tuple size.

```
virtual out rate= sum(input rate, VIRTUAL CHILDREN)
node out rate= sum(virtual out rate, VIRTUAL NODES)
             +sum(operator out rate, OPERATORS)
system cost = avg(node out rate, BROKERS)
```

Query-oriented metrics. This new set of metrics expresses performance criteria for the operators and queries in our system. At the first level, we define the cost of an operator by aggregating local metrics collected from neighbors of the current operator's location. These neighbors can be the nodes connecting the operator to the location of (i) its upstream operator in a query plan (referred to as *upstream aggregation*) or (ii) its immediate downstream operator (*downstream aggregation*). If the operator has multiple upstream or downstream operators across multiple query plans, we define the cost of each one independently and average the costs to compute the final cost of the operator. Note that nodes hosting operators are responsible for collecting the local metrics from their neighbors. At the second level, we define the cost of a query by aggregating the cost of all operators. Finally, the system cost is computed by aggregating the costs of all queries.

An example metric for upstream aggregation is the operator latency. In this case, an operator's cost can be computed as the latency to its upstream operator plus its processing latency. The latency of a query can then be computed as the sum of the costs of all its constituent operators. Thus, for the average query latency we have:

```
operator latency= sum(latency, UPSTREAM OPERATOR)
                + processing_latency
query latency= sum(operator latency, OPERATORS)
system cost= avg(query latency, QUERIES)
```

To implement this metric, we define the processing latency of an operator as a local metric and provide a function for its estimation. Moreover, each virtual node is customized to measure the latency to its parent and collect the link latency of the nodes on the path to its upstream operator in the query plan. To illustrate this, we consider the query distribution in Figure 1. In this case, n_4, the host of A_2, will measure its

latency to n_3 and collect the latency between n_3 and n_2 and n_2 and n_1 from n_3 and n_2 respectively.

Our cost model allows for some levels of aggregation to be omitted and node and query oriented metrics to be combined. An example of a metric with one level of aggregation is the maximum processing load across nodes. The local metric of each node is its processing load, *i.e.*, the sum of its local operators' load. We define the load of an operator (in % of CPU cycles per time unit) as the product $r \times s \times c$, where r is its input rate, s its selectivity and c its cost (% of CPU cycles per tuple). Hence, the maximum load is defined as:

```
node load= sum(load, OPERATORS)
system cost= max(node load, BROKERS)
```

Constraints. We allow applications to define constraints on performance metrics, using the same multi-level aggregation model. Constrained metrics can be defined with respect to nodes and virtual nodes, operators or even the whole set of nodes or queries. For example, application designers may impose an upper bound on the number of immediate children of each virtual node or on the maximum processing load across brokers. Moreover, they can limit the number of replicas an operator may have, the number of operators a node can migrate, or require an upper bound on the maximum query latency. XFlow customizes its functionality and optimization to respect these constraints.

We inherit from XPORT two types of aggregation functions for defining the operator, query and system cost (or virtual, node and system cost): (i) *bottleneck* (MIN, MAX) and (ii) *additive* (SUM, AVG). These functions permit applications to define a variety of commonly used system cost measures, like maximum load, bandwidth consumption, maximum query latency, average query load, etc.

4. DISTRIBUTED OPTIMIZATION

XFlow distributes the query operators, aiming to minimize the global system cost. It continuously refines the placement of the operators and their execution, through migration, replication and partition operations. We collectively refer to these as *operator distribution operations*.

Our optimization framework strives to meet three requirements. First, it must be *scalable* in terms of the number of operators and the number of brokers in our network. Second, our operations should be *efficient* and *adaptive* to time-varying network or workload conditions. Finally, we require a *metric-independent* model that uniformly applies to and handles a variety of application-defined cost metrics.

To address these challenges, we designed a general, decentralized framework that does not rely on global information and has low communication overhead. Our protocol continuously considers *localized distribution operations* on the running operators. Each node maintains information about its own "neighborhood" and periodically attempts to distribute its local operators across the nodes within the scope of this neighborhood. An advantage of this approach is that non-overlapping neighborhoods can be optimized locally and concurrently. XFlow exploits its structured cost-definition model and the known semantics of the aggregation functions to derive generic properties and equations to quantify the benefits of any candidate distribution operation.

4.1 Operator distribution

XFlow nodes distribute their operators across their neighbors in the overlay trees they participate it. To identify the scope of this "logical" neighborhood, we rely on the notion of an *optimization unit* as defined in [23]. An optimization unit of node n is defined over a single tree and is the set of nodes in a k-level subtree rooted at n in this tree, where k is a system parameter. In XFlow, each node owns such a unit for each tree it participates in. Hence, there is a unit for every virtual node in our system. We define a node's neighborhood to be the union of all optimization units it owns. In our current implementation, each unit has at most three levels, including the owner, its parent and its children in the tree. We denote as n_i^k a virtual node of node n_i in tree T_k, and in Figure 4(a) we show two optimization units of n_i in two trees, T_1 and T_2.

We distribute the operators with two types of operations: (i) *operator placement operations*, which migrate operators to alternative locations and (ii) *operator execution operations*, which change the implementation of operators by replicating or partitioning them across multiple nodes.

4.1.1 Operator Placement and Migration

Operator placement modifies where an operator is executed. For a given operator, we reduce the space of the candidate network locations by considering only neighbors of its current host. These are the nodes in the optimization units the node owns across the multiple trees it is a member of. Hence, our algorithm checks the following placement cases in these trees: (i) the parent of the host, (ii) its children and (iii) its siblings. Although at each step we consider a small number of alternative locations, we can gradually migrate operators to a different network area than that of their original hosts. As an example, if improving the query latency is our performance goal, operators are continuously placed one level higher in the tree, *i.e.*, closer to their upstream operators, as that would reduce the network latency of their input tuples. Moreover, if an application aims to reduce bandwidth consumption, operators with selectivity less than one (*e.g.*, filters) migrate closer to their upstream operators, while if the selectivity is more than one (*e.g.*, joins), they are pushed closer to the downstream operators. This process reduces the overhead of forwarding tuples to the network.

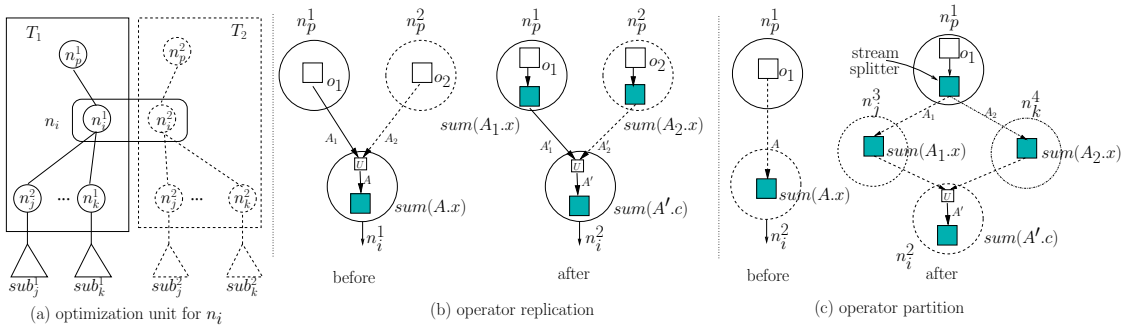
We exploit our underlying pub-sub model to dynamically reroute data flows to the new locations of the operators. More specifically, the new host of the operator subscribes to the inputs of the operator by joining the trees that publish these streams (if not already part of the them). If the operator is also an internal source, then the new host publishes the operator's output to the root of the corresponding tree.

4.1.2 Operator Replication and Partitioning

The primary goal of replication and partitioning is to parallelize operator execution to utilize idle resources available in the broker network.

Operator Replication. Replication is applied to operators that subscribe to multiple trees. Instead of collecting all streams and executing the operator on a single node, replication attempts to exploit processing resources of multiple trees. The goal is to process each input stream independently on each tree and combine the results to construct the final output. To implement this, a replica is created for each of the trees the operator receives input from. Each replica will receive one of the inputs of the original operator. To guarantee correctness, a *final operator*, receives the outputs of all replicas and produces the final result.

In our current implementation, we handle replication on



(a) optimization unit for n_i (b) operator replication (c) operator partition

Figure 4: Optimization units of n_i and operator replication and partition.

filters and sum, count, min and max aggregation operators. Depending on the semantics of the replicated operator, we need to use a different implementation for the final operator. For example, for filters we merge the already filtered outputs of our replicas using a union operator. For the case of min and max operators, we apply the same min or max operation on the replicas outputs, while for the sum or count operators, the final operator sums the replicas' results.

Operator replication for the sum operator is shown in Figure 4(b). The original operator receives data from two streams A_1 and A_2, after a union operator merges them into stream A. We sum the attribute x, which exists on both streams A_1 and A_2. After replication, we apply the same operator independently on A_1 and A_2, and carry the results in the c attribute of their replicas' outputs A'_1 and A'_2. We sum the c attribute from both streams (after we merge them with a union operator) in order to get the final result.

The benefits of replication primarily depend on the location of the replicas. For example, if improving the query latency is our objective, then placing the replicas in Figure 4(b) on the same node as the original operator will not decrease latency. To increase the replication's benefit, we combined this operation with a migration operation: nodes can migrate replica to one of the nodes across all their optimization units. Since each replica corresponds to a different tree, a node can place the replica on its neighbors on this tree, *i.e.*, its parent, one of its children or one of its siblings. Note that we do not require *all* replicas of an operator to be migrated, only the ones that can further improve performance. The pub-sub network easily routes stream flows from the replicas: they publish their outputs through a tree, and the final operator's host subscribes to them by joining these trees.

Replication can also increase the opportunities for optimization. Replicas are single input operators that can be handled independently by our framework. Thus, they are more flexible to migrate as they affect fewer nodes and trees than operators with multiple inputs and multiple subscriptions. For example, each replica can be migrated closer to its own data source, reducing the network latency of queries.

Operator Partition. Partition is applied on the input stream of an operator. It splits the stream into two flows and each sub-flow is processed by a different replica of the operator, thereby exploiting data parallelism when possible. The output of the replicas are processed by a final operator which produces the final results. Its type depends on the original operator, following the same rules as for the case of replication. Similarly to replication, we combine partition and migration into a single operation and place the replicas to a neighbor of the original host. The hosts of the replicas join the trees providing their inputs and the pub-sub network routes the output streams published by the replicas to the location of the final operator.

Operator partition is shown in Figure 4(c) for the sum operator. Node n_p^1 splits stream A and publishes two sub-streams A_1 and A_2. These streams are routed to the two replicas and their outputs are merged before being processed by the final operator. For this example, we assume that node n_i participates in trees T_3 and T_4 and it places the replicas on some instances of its neighbors n_j and n_k on these trees.

Partition can be used to reduce the processing requirements of the original host and move a portion of the processing to another node. Each replica is an independent operator that processes half of the initial stream. Moreover, assuming a selectivity less than one for the replicas, the incoming rate of the final operator is also lower than that of the initial stream, thus the processing overhead of its host node is decreased.

4.2 Evaluating cost improvement

Our optimization framework includes a generic cost model that quantifies the expected benefit of an operation with low overhead. We exploit the fact that our localized operations affect only a subset of the network brokers and operators, and evaluate the impact on the cost metrics of only the affected entities. For the purposes of illustration , we will describe our cost model with respect to the query-oriented metrics and the SUM and MIN aggregation functions. Similar results can be obtained for the node-oriented metrics and the AVG and MAX functions in a straightforward manner.

We start by providing the dependencies among nodes, operators and queries. We assume a set of queries Q. Each query $q_i \in Q$ consists of a set of operators and let O denote the total set of operators. Let also E_i be the set of queries that include operator $o_i \in O$.

We denote the local metric of node n_i as l_i. Any operation that involves this node could affect its local metric (*e.g.*, adding an operator could increase its processing load). We denote such changes as $\Delta(n_i : l_i \rightarrow l_i + \delta)$. The cost of an operator $o_i \in O$, oc_i, is defined by aggregating the local metrics of some neighbors (*e.g.*, latency of the nodes to the upstream operator). This cost could change due to changes on the local metrics (*e.g.*, increase on the link latency) or changes on the set of neighbors (*e.g.*, placing the operator on another node.) The following definition identifies which operators and queries (that do not reside on this node) may be affected by a change on a node's local metric.

Definition 1. (DEPENDENT OPERATORS) *The* dependent operators *of node n_i is the set of operators, D_i, whose*

Symbol	Definition
l_i	local metric of n_i
oc_i	cost of operator o_i
qc_i	cost of query q_i
O_i	operators executed on n_i
Z_i	the set of queries including operators in O_i
E_i	the set of queries including operator o_i
D_i	dependent operators of node n_i
G_i	dependent queries of node n_i
F_α	affected nodes from operation α
b_α	benefit of operation α

Table 1: Model Terminology

cost metrics depend on n_i's local metric, l_i:

1. *If the cost of an operator is defined as an upstream aggregation, then D_i is the set of the first operators reached by n_i's paths to the leaves of every tree n_i participates and every tree in which it publishes data.*
2. *If the cost of an operator is defined as a downstream aggregation, then D_i is the set of first operators reached by n_i's path to the root of every tree n_i participates.*

DEFINITION 2. (DEPENDENT QUERIES) *The* dependent queries *of n_i is the set of queries, G_i, that include at least one dependent operator of n_i. In particular: $G_i = \cap_{o_j \in D_i} E_j$.*

Let us consider the query plan and its network distribution in Figure 5. We assume that the cost of an operator is its latency defined as an upstream aggregation. Then a dependent operator of n_2 is o_4, as any changes on n_2's latency to the host of o_3, n_3, could affect the output latency of operator o_4.

Each distribution operation directly affects a set of nodes. Operator migration has an impact on the origin and destination nodes, while replication and partition affect the origin node and the nodes where the replicas are placed. Let F_α be the set of affected nodes for an operation α. Changes on a node $n_i \in F_\alpha$ could change the cost of every dependent query as well as the cost of queries that include any operator executed on these nodes. Let O_i be the set of operators executed on n_i and Z_i the set of queries including an operator in O_i.

XFlow's cost model does not evaluate the new cost of every dependent query, but instead quantifies the impact of their cost changes on the global system cost. This impact is expressed through the *aggregated dependence cost*.

DEFINITION 3. (QUERY DEPENDENCE COST). *The* query dependence cost, *$c(G_i)$, of a query dependent set G_i, is the aggregation of the cost of every query $q_j \in G_i$, using the aggregation function that defines the system cost.*

For example, if the query cost and the system cost are defined based on the SUM function (*i.e.*, we aggregate operators and queries with the SUM function), then:

$$c(G_i) = \sum_{q_m \in G_i} qc_m = \sum_{o_j \in D_i} \sum_{m \in E_j} qc_m. \quad (1)$$

The following property identifies the effect of an operation on the system cost, based on the query dependence cost.

System cost-effect property. *Assume the system cost is defined based on the* SUM *function. Given an operator distribution operation α, the expected change of the system's performance is the following*:

$$b_\alpha = \sum_{n_i \in F_\alpha} (\sum_{q_m \in Z_i} \Delta qc_m + \Delta c(G_i)). \quad (2)$$

If the system cost is defined based on the MIN *function, then:*

$$b_\alpha = \min_{n_i \in F_\alpha} \{ \min_{q_m \in Z_i} \{qc_m + \Delta qc_m\}, c(G_i) + \Delta c(G_i), \min_{q_m \notin Z_i, q_m \notin G_i} \{qc_m\}\} - c \quad (3)$$

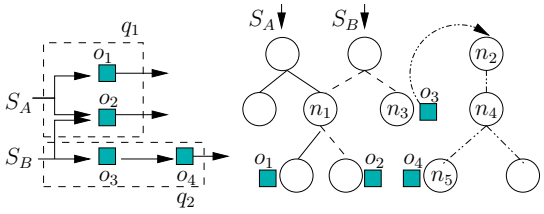


Figure 5: Example of operator distribution.

In the following section we discuss how we can evaluate the above equations.

4.2.1 Query cost changes

We begin this section by evaluating the change on the dependent operators of a node and continue with the derivation of the change on query costs and the query dependence cost. We first focus on the case where the operator cost is defined based on the MIN function (*i.e.*, we aggregate local metrics of neighbors using the MIN function). In this case, whether a node can affect an operator's cost depends on the rest of the nodes on the path to the upstream (or downstream operator). We capture this in the following definition.

DEFINITION 4. (CRITICAL VALUE) *Let n_i be a node. If the cost of operators is defined based on the* MIN *function, then the* critical value *of a dependent operator $o_j \in D_i$, w.r.t. n_i, $h_i(o_j)$, is the minimum local metric of all nodes on the path between n_i and the current location of o_j.*

To explain this, we use the example in Figure 5, and let us assume that the cost of an operator is the minimum latency to its upstream operator. This is an upstream aggregation, thus n_2 has o_4 as its dependent. The local metric of each node is the latency to its parent and $h_2(o_4)$ is the minimum latency of all nodes connecting n_2 and n_5.

Operator cost-effect property. *Assume a change $\Delta(n_i : l_i \rightarrow l_i + \delta)$ that triggers a change Δoc_j on the cost of a dependent operator $o_j \in D_i$. If the operator cost is defined based on the* SUM *function, then $\Delta oc_j = \delta$, while if it is defined based on the* MIN *function, then $\Delta oc_j = \lambda$, where*

$$\lambda = \min\{l_i + \delta, h_i(j)\} - \min\{l_i, h_i(j)\}.$$

To explain the above, we assume in Figure 5 that the operator's cost is its output latency. Hence, we aggregate the link latencies between nodes n_5 and n_3 to get the latency of operator o_4. If the latency between n_4 and n_2 increases, then the latency of o_4 will have the same increase. If the operator cost is the minimum latency to its upstream operators, then the cost of o_4 is the minimum latency of the links between n_5 and n_3. A change on the latency between n_4 and n_2, will change the cost of o_4 by λ. The λ parameter identifies the difference between the new minimum latency between nodes n_5 and n_3 and their current latency. This parameter can be further simplified under certain conditions, shown in Table 2.

Changes on the cost of an operator $o_i \in O$ will affect the cost, qc_j, of any query $q_j \in E_i$. The following property describes how changes on the cost of operators can be translated to changes on the queries' costs.

Query cost-effect property. *Assume a change $\Delta(o_i : oc_i \rightarrow oc_i + \delta)$ that triggers a change Δqc_j on the cost of a query $q_j \in E_i$. If the cost of a query is defined based on the* SUM *function, then $\Delta qc_j = \delta$ and if is defined based on the* MIN *function, then $\Delta qc_j = \tau$, where $\tau = \min\{\beta_i(j), (oc_i + \delta)\} - qc_j$ and $\beta_i(j) = \min_{o_m \in P_j, o_m \neq o_i}\{oc_m\}$.*

l_i change	Conditions	λ
$\delta < 0$	$\min_{o_j \in D_i}\{h_i(j)\} \geq l_i$	δ
$\delta > 0$	$\min_{o_j \in D_i}\{h_i(j)\} \geq l_i$	δ
$\delta > 0$	$h_i(j) \geq (l_i + \delta)$	δ

Table 2: Change on operator cost oc_j upon change on local metric l_i (if oc_j is defined based on MIN function).

Assume the query cost is the sum of its operators' latencies. Then, in Figure 5, decreasing the latency of o_2 will decrease the cost of q_1 and q_2. If the query cost is defined as the minimum latency of all its operators, then decreasing o_2's latency might create a new cost for q_2, depending on o_4's latency. The τ parameter takes into account the minimum cost of all query operators except o_2 (that is value $\beta_i(j)$) and identifies the change on the query cost. Once we have evaluated the cost changes on dependent queries, we evaluate the new value for the query dependence cost, based on Definition 3.

Cost evaluation steps. The above properties allow us to estimate an operation's benefit. First, we evaluate the change on the cost of migrated operators and replicas by aggregating the local metrics for their new neighbors. The aggregation function and the set of neighbors depend on the definition of the operator cost (upstream or downstream aggregation). Given a change on the operator cost, we quantify the impact on their query costs based on the query effect property.

In the next step, we estimate any changes on the local values of the affected nodes, based on the definition of the local metric. Since the local metric is a function on local statistics, we evaluate the new local metric based on the local statistics of the affected nodes. For example, if the local metric is the node's processing load, and we migrated an operator to the node, then we add on the destination node's load the expected processing load of that operator, as estimated by its current host's statistics. Depending on these changes, our protocol uses the operator and query cost-effect properties to evaluate the impact on the dependent queries of these nodes and the new query dependence cost. Given this, we use Equation 2 or 3 to evaluate the benefit of the operation.

4.3 Operator distribution protocol

The optimization process proceeds over *optimization periods*. At each period, each node collects statistics from the candidate locations for the operators and their replicas and tries to refine the placement or execution of its local operators. It iterates overs its local operators, and evaluates the benefit of each operation it can apply on them. The most effective of all pairs (operator, operation) is selected and sent to the root of the tree. The roots of all trees collaboratively identify the most beneficial operation and inform the host of the operator about the operation it should apply. This approach ensures cost improvement in every optimization period, assuming that at least one beneficial operation has been identified.

Dynamic operator modifications. Our system adopts the "pause-drain-resume" approach to migrate or change the execution of stateless operators. When a node decides to modify an operator, its pauses the data flow to that operator, and starts buffering any incoming tuples. The operator executes any remaining tuples and after the operation is applied the node resumes the data flow. To handle migration of stateful operators we adopt solutions proposed in the literature [29].

Optimization state. Table 3 shows the state nodes need to maintain for the different functions for aggregating the operator and query costs. This state is derived based on Equations 2 and 3 and the operator and query cost-effect properties. For example, to use the operator effect property for the MIN function, nodes need to know the critical value of their local and dependent operators. By maintaining this state we reduce the communication overhead during the optimization process. Note that this state is common for any operation of our framework and independent of the actual performance metric. Moreover, it depends in most cases on the dependent entities of a node and not on the global set of nodes, queries or operators in the system.

Operator Aggr	**SUM**	**MIN**
Query Aggr		
SUM	size of Z_i size of G_i	critical value $\forall o_j \in O_i$ critical value $\forall o_j \in D_i$ size of E_j $\forall o_j \in D_i$ dependence cost $c(G_i)$
MIN	system cost, dependence cost $c(G_i)$ value $\beta_i(j)$ $\forall q_j \in Z_i$ B_j $\forall o_j \in D_i$	system cost dependence cost $c(G_i)$ critical value $\forall o_j \in O_i$ value $\beta_i(j)$ $\forall q_j \in Z_i$ critical value $\forall o_j \in D_i$ B_j $\forall o_j \in D_i$

Table 3: Optimization state for n_i and $B_j = \{(\beta_j(m), oc_j) | q_m \in E_j\}$.

The query manager of a node n_i maintains also information about the set of local operators, *e.g.*, their cost and the root brokers publishing their inputs. This operator-related state has size $O(|O_i|)$. Nodes also need some information about the trees they participate, *e.g.* the cost of each virtual node they own, its profile in each tree and the profiles of each of its children per tree. This state is of size $O(|T_i|)$, where T_i is the set of trees in which n_i participates.

Optimization traffic. During optimization, nodes exchange statistics with their neighbors in the optimization unit. This is the information required from our protocol to evaluate the changes on the local metrics of the affected nodes and on the operators to be migrated, replicated or partitioned. This state depends on the definition of our cost metrics. For example, for the query latency metrics defined in Section 3, nodes need to know the latency of the candidate locations from the upstream operators. Thus, they collect the local metrics from the nodes connecting the candidate location to the location of the upstream operators.

Periodically, nodes exchange data in order to calculate their local state. To minimize this maintenance traffic, nodes gather and calculate this information in a hierarchical fashion. They aggregate the state from their children and push the result to their parents across the trees. For example, to maintain the size of the downstream dependent operators, nodes aggregate the number of operators in their subtrees and forward the sum to their parents.

Batch-executing multiple optimization steps. Although applying a single operation per optimization period will improve performance over time, considering multiple operations could speed up convergence to better configurations. To this end, we allow multiple operations to be applied during a single period, *e.g.*, migrating multiple operators.

We use a standard best-first-search-like algorithm for identifying an effective set of operations. At each step, we consider all possible operations for each operator on the node and evaluate their benefits. The best operation that does not violate any constraint is selected, and, given the new configuration defined by this operation, we reevaluate the benefit of distributing another operator from the remaining ones. We continue by picking the best combination of the (now two)

operations, continue this best-first search iteratively, and stop after a tunable but fixed number of steps.

Benefits of the pub-sub model. Using a pub-sub model for disseminating data flows has a number of advantages for our optimization. First, the loosely-coupled architecture of pub-sub networks allows them to deal efficiently with churn. XFlow treats our operators as consumers and subscribers and thus, our optimization operations are handled as subscriptions/unsubscriptions from the trees. This allows XFlow to be scalable and robust in terms of the number of dynamic operator changes.

Second, it allows for dynamic creation of multiple trees, depending on the degree of sharing of external and internal sources. Clients can join trees that carry only relevant data, reducing redundant transmissions, while multiple large trees improve the optimization process, as they increase the neighborhoods over which we can distribute operators. High degree of sharing has been shown in practice for external [20] and internal sources [4]. In the case of low inputs sharing, grouping input streams will allows to get the similar benefits.

5. PERFORMANCE EVALUATION

We have implemented an initial prototype of XFlow in Java and studied the performance of our system in deployment on the PlanetLab testbed. In our experiments, we used a network of up to 150 PlanetLab sites and up to 900 queries. The input streams for each query are chosen from a set of 100 real-world input streams using a Zipf distribution. Each input stream is an RSS feed published from an RSS source with an average feed size of 6.3KB. To emulate the behavior of and easily experiment with a range of operator types, we adopted a black-box approach when modeling operators, representing an operator abstractly using (i) a processing cost parameter (uniformly distributed in [0,1] to indicate the fraction of the available CPU cycles it requires per unit time) and (ii) a selectivity parameter (uniformly distributed in [0,1]).

The external RSS sources are assigned randomly to a set of four root brokers, while the clients are hosted by the remaining brokers. Grouping sources into four trees, allows more nodes to connect to the same tree, leading to larger neighborhoods over which operators can be distributed. We assign clients to brokers randomly, when no constraints are defined. Otherwise, we use an assignment that respects the constraints. Depending on the inputs of their local queries, nodes connect to the proper tree, by picking a random node from the ones already in the tree as their parent.

We used our prototype to implement three distributions of operators, each one optimizing a different metric. These metrics are: (i) average query latency, (ii) maximum processing load across all nodes and (iii) total bandwidth consumption (defined in Section 3). Our experiments demonstrate XFlow's flexibility and effectiveness, as it manages to improve these metrics significantly over a sequence of optimization steps.

5.1 Operator placement

We start our discussion by demonstrating the efficiency of our operator placement approach. For simplicity, we here consider queries consisting of single operators, each operating over a single input stream. We examined four alternative placement approaches. CLIENT-STAR assigns operators to the location of their client and their hosts connect directly to the root brokers, creating a star topology for every tree.

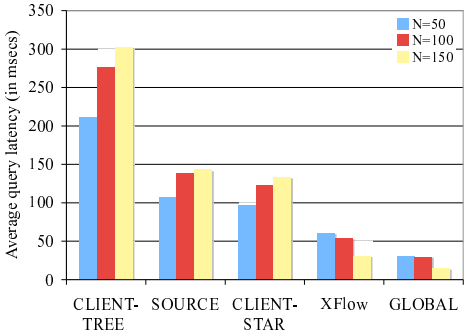


Figure 6: Average query latency on PlanetLab for 500 queries. XFlow performs better than all practical approaches and approximates GLOBAL. (N is the network size.)

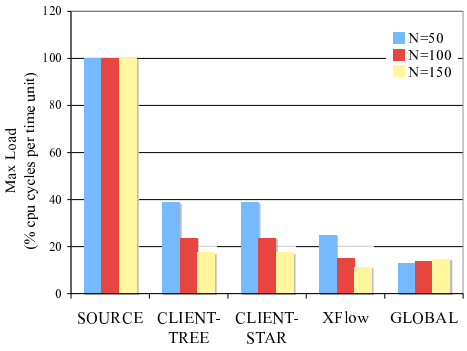


Figure 7: Maximum processing load for 500 queries. XFlow performs close to the GLOBAL approach.

CLIENT-TREE also assigns the operator to the host of the client, however, nodes connect to the trees through a random member of the tree. SOURCE places operators to the root brokers publishing their input stream. The root brokers process the queries and forward the results to the clients. Placing operators on the edge of the network was an approach studied also in [8]. GLOBAL applies a greedy strategy that considers all existing queries in the order they were registered to the system and places each after an exhaustive search over all possible placements. This requires global knowledge of the query set and workload, and is not infeasible in practice, but it gives a target upper bound for the performance of our algorithms. For each metric, we used a different implementation of GLOBAL.

We compared the alternative placement approaches with XFlow's protocol. Our experiments reveal that, depending on the optimization metric, different approaches achieve the best placement. XFlow consistently performed very close to the best placement regardless of the performance criteria used.

Latency. Figure 6 shows the average query output latency for different network sizes and placement algorithms. This latency includes the network latency as well as the processing overhead. The more operators a node must process and the higher its fanout in the tree, the greater the processing latency that feeds observe when traveling through that node. For the GLOBAL approach, we assign operators to the node with the lowest latency to the root broker publishing their input streams. To reduce further the network latency, we assign clients to the same node as their queries, so that processing results are not forwarded in the network.

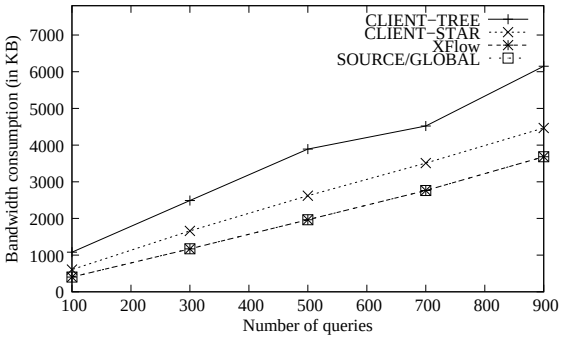


Figure 8: Total bandwidth consumption for 100 Nodes. XFlow continuously refines its operator placement and converges to SOURCE's performance.

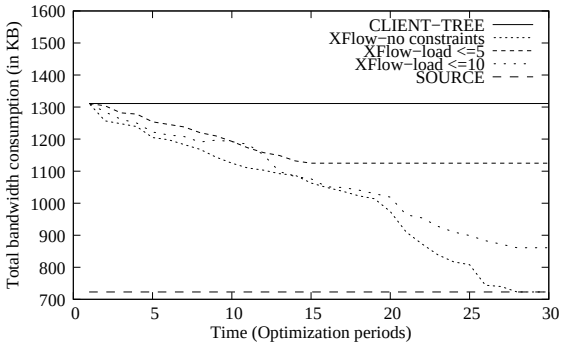


Figure 9: Total bandwidth consumption under constraints on the processing load of each node. The performance improves, but it is limited by the constraints.

CLIENT-TREE has the highest latency as queries receive their inputs through random paths to the roots, possibly with multiple network hops. The latency increases with the network size, as more nodes participate and the random trees tend to have higher height. SOURCE and CLIENT-STAR both create a star topology. Hence, the processing latency at the roots of the trees is high, since they have to filter the incoming tuples for each of their children in the trees. For SOURCE, the performance is worse, since the roots have to process all the operators as well.

GLOBAL has the best performance for 150 nodes because the node with the least latency to the root brokers was not included in the set of 50 and 100 nodes. XFlow performs very close to GLOBAL. XFlow incrementally migrates operators and significantly improves the performance by placing operators on brokers with smaller network and processing latency. Our protocol applies around 160 migration operators, replacing an average of 17 operators in a each operation. Our performance is comparable to GLOBAL and significantly outperforms all the other practical approaches.

Processing Load. We also studied the maximum processing load across all nodes (Figure 7). For this metric, placing all operators on the root brokers (SOURCE) leads to worse performance, as it utilizes all the resources from the root brokers. CLIENT-TREE and CLIENT-STAR distribute the load across all nodes that host a client. However, they perform worse than XFlow and GLOBAL. The GLOBAL algorithm places every new query at the least loaded node. XFlow applies around 42 migrations and performs near as well as GLOBAL. As network size increases, it performs even better, since more nodes are available for distributing the processing. In the case of 150 nodes, XFlow outperforms GLOBAL.

Bandwidth. Figure 8 shows the total bandwidth consumption for all approaches and varying number of queries. The best performance in this case is achieved by placing operators with selectivity less than one close to the sources. This approach eliminates input tuples close to their sources, reducing the amount of data forwarded in the network. Thus, both SOURCE and GLOBAL place the operators on the root brokers and achieve the best performance. CLIENT-STAR consumes more bandwidth as all input tuples are forwarded to the clients for processing. CLIENT-TREE performs worse, because input tuples are forwarded to their clients through multiple hops. However, XFlow manages to continuously refine the operator placement and converge to a distribution that requires low bandwidth consumption, *i.e.*, over time, it moves almost all operators to the root brokers, performing the same as GLOBAL.

Constrained metrics. One way of achieving the benefits of pushing all operators closer to the sources, without saturating the processing resources of the root brokers and the nodes close to them, is by adding constraints on the maximum load of the nodes. These constraints will prevent operators from migrating to the root brokers. We expressed such constraints and studied XFlow's bandwidth consumption on 100 PlanetLab sites using 200 queries.

Figure 9 shows the total bandwidth consumption when we set the load constraints to 5% and 10% of the node's CPU cycles and when no constraints exist. Our system incrementally improves its performance in all cases. In the absence of constraints, all operators end up being executed at the root brokers. However, when constraints exist, many operators cannot move to higher levels, as this would lead to a violation of the load constraint.

Effect of initial placement. We use the maximum load metric to study the effect of the initial placement of operators. Figure 10 shows the improvements we can achieve compared with the system's performance when we initially place the operators (i) on the clients' hosts (XFlow-Client), (ii) on the root brokers (XFlow-Root) and (iii) on the least loaded node (XFlow-Greedy). XFlow significantly improves the maximum load for all cases. The improvement is higher for the XFlow-Root case since maximum load for the initial performance is already high (see Figure 7). Our protocol migrates as many operators as possible from the root brokers and distributes the processing overhead across the network. XFlow-Client and XFlow-Greedy demonstrate smaller improvements as we already start with a good initial placement. However, even in the case of XFlow-Greedy, we manage to improve the performance by 28% when the network size increases to 150 nodes, by utilizing better the available nodes in the system.

5.2 Operator execution operations

We now study our operator execution rules. We start by examining replication and then focus on operator partitioning. We allow our operators to be replicated or partitioned only once. In these experiments, we used a workload of 200 queries, each consisting of one operator, where each operator requests input from two different root brokers. As a result, each node executing an operator must subscribe to two trees.

Replication. Figure 11 shows the improvement on bandwidth consumption when replication is used. We compare

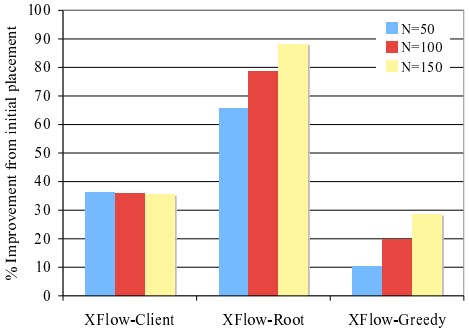


Figure 10: Improvement of maximum load for different initial placements. Performance is significantly improved for all cases.

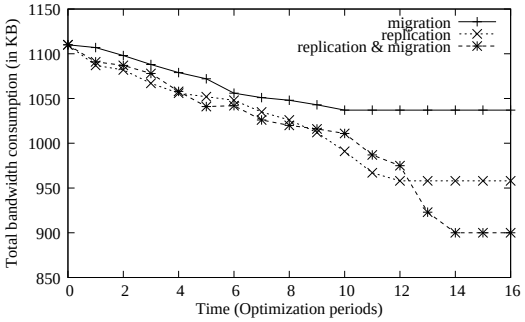


Figure 11: Total bandwidth consumption when replication is applied. Combining replication and migration can lead to better performance.

three cases in which we apply: (i) operator migration, (ii) operator replication, and (iii) migration and replication together. The improvement achieved with migration is very limited, since moving an operator to another node in one tree might not yield any benefit, as this node may not be subscribed to the second tree. When only replication is used, the performance is better. However, since no migration rules are allowed, these replicas are not reallocated and the bandwidth consumption can not improve further. Not surprisingly, best results are obtained when replication and migration are used together. In this case, our optimization process will try to migrate our replicas closer to the sources, if this reduces the data forwarding in the tree. Since these replicas take input from a single tree, it is easier to identify a beneficial migration.

Operator partition. Figure 12 shows similar results for the operator partition case. In this experiment we focus on improving the maximum load of the network. Migrating an operator can have some benefits, however, since the load of a query depends on its input rate, reducing this rate will reduce its load as well. Partitioning an operator can achieve this, as now half of the input rate is processed by each replica. Moreover, when migration is allowed, these replicas can move independently in their respective trees, independently utilizing more processing resources and improving performance.

5.3 Optimization Overhead

At each optimization period, each node collects data from the other nodes in its neighborhood in order to estimate the *benefits* of candidate optimization operations. We refer to this traffic as the *optimization traffic* (see Section 4.3). Intuitively, the optimization traffic is a measure of the optimization overhead. Moreover, in order for nodes to maintain their optimization state (see Table 3), they periodically exchange data, leading to *maintenance traffic*. In our current implementation, nodes are set to exchange this data every second but this rate can be much coarser in a real deployment. We run experiments on 150 PlanetLab nodes and 500 profiles for the three metrics we implemented. Figure 13 shows the average optimization and maintenance data for each node. The results reveal that our protocol had very small communication overhead, as it does not exceed 200Bytes in the worst case.

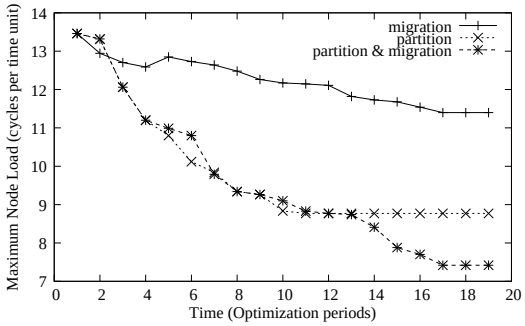


Figure 12: Maximum load across nodes when partition is applied. Combining migration and partition can lead to better performance.

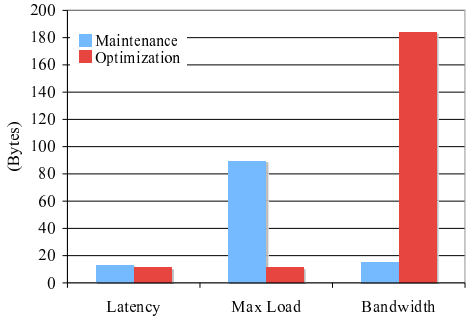


Figure 13: Network traffic for different metrics.

More specifically, for the query latency, each virtual node periodically sends its latency from the root of the tree to its children, in order for them to evaluate their own latency. Similarly for bandwidth, virtual nodes send their input rates to their parent in the tree. In the case of maximum load, nodes calculate their processing cost locally. However, data is exchanged in order for nodes to evaluate their local state, *e.g.*, the maximum load of their subtree and the maximum load in the network apart from their subtree.

During optimization, for the query latency metric, nodes collect only the latency of all candidate locations for their operators or replicas; while for the maximum load metric, nodes need to know the current load of the candidate locations. Hence, the traffic for these cases is low. However, for the bandwidth consumption metric, nodes exchange the profiles of their neighbors, the profiles of their children, and statistics on the input rates of their streams. This information is required in order for a node to evaluate how the output rate of the candidate location will be affected.

6. RELATED WORK

Much work has focused on content-based pub-sub systems including *e.g.*, SIFT [27] for text documents, ONYX [16] and

XRoute [14] for XML messages and SIENA [11] and SemCast [22] for relational data. These systems focus on either reducing the bandwidth usage [14, 22] or improving the efficiency of profile matching at each node [11, 13, 15]. Moreover, they commonly support subscriptions with only predicate-based filters over individual events, so they cannot express complex stream-processing queries across multiple messages and streams. The SASE [26] event processing system extends the pub-sub model to allow the execution of stateful event processing queries. Among other differences, SASE is a centralized system and addresses neither operator placement in wide-area networks nor extensibility.

Our work is directly relevant to distributed stream processing, for which several proposals have emerged in recent years. In general, these systems are designed, optimized and evaluated in small-scale cluster environments. As such, they do not address scalable and adaptive data collection and distribution. They also lack extensibility in terms of the data types, optimization techniques and metrics they support. As a case in point, Borealis [6] supports run-time operator migration but not operator replication or partitioning. Another example is Medusa [9], which is a similar system that performs dynamic load distribution in a federated setting based on agoric principles. Furthermore, neither Borealis nor Medusa addresses overlay network management, which we believe is critical to ensure network scalability and adaptivity.

Network-based querying and dissemination systems are also related to our work. PIER [17] aims to build an Internet-scale querying system on top of a routing network given by a DHT, where operators are placed randomly. The operator placement problem has been studied in SAND [8], SBON [24] and [25]. SAND and SBON focus on minimizing the bandwidth usage, while [25] assumes a hierarchical stream collection architecture which is not applied in our architecture. Very recently, COSMOS [28] combined the pub-sub model with stream processing in order to improve load balancing among nodes and minimize the communication cost. None of these approaches addresses adaptive query distribution and allow for operators to be replicated or partitioned. Moreover, they lack the generality and extensibility of XFlow, as they focus on specific optimization metrics.

7. CONCLUSIONS AND FUTURE WORK

Sophisticated Internet-scale Monitoring (ISM) applications are fast emerging. We proposed XFlow as a distributed extensible infrastructure system to effectively support a variety of ISM applications. The key innovations of XFlow include (i) a novel architecture that tightly integrates stream processing and pub-sub, (ii) a generic optimization framework that can express a variety of query and node-oriented optimization criteria, and (iii) a low-overhead run-time system that creates, maintains, and optimizes a multi-tree overlay network. Our experimental results, based on an early XFlow prototype, shows that the system is viable and effective.

This paper presents an initial step towards a robust ISM system. There are several areas for immediate exploration and extension. First, we would like to implement a real ISM application using XFlow and deploy it as a public service on PlanetLab. This experience will allow us to better debug our system, gather real user profiles and usage patterns. Second, we would like to extend our operations to include tree-centric optimizations (to enable tree merges, splits, etc.) and operations that affect the quality of query results (*e.g.*, adding a load shedding operator). Finally, we would like to provide better support for user defined functions, which will likely occur frequently in ISM applications that involve rich data types. We believe we can treat and optimize them as well as we do our built-in operators, whose semantics are well-known, by providing a narrow "hinting" interface which the user can use to specify relevant properties of the operator (*e.g.*, whether/how the operator can be parallelized).

8. 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] Adler et al. Channelization problem in large scale data dissemination. In *ICNP*, 2001.
[8] Y. Ahmad and U. Cetintemel. Network-aware query processing for stream-based applications. In *VLDB*, 2004.
[9] Balazinska et al. Contract-based load management in federated distributed systems. In *SIGMOD*, 2005.
[10] Campbell et al. IrisNet: an internet-scale architecture for multimedia sensors. In *MM*, 2005.
[11] A. Carzaniga and A. L. Wolf. Forwarding in a content-based network. In *SIGCOMM*, 2003.
[12] Carzaniga et al. Design and evaluation of a wide-area event notification service. *ACM TOCS*, 19(3), 2001.
[13] Chan et al. Efficient filtering of XML documents with xpath expressions. In *ICDE*, 2002.
[14] R. Chand and P. Felber. Scalable protocol for content-based routing in overlay networks. In *NCA*, 2003.
[15] Y. Diao and M. J. Franklin. Query processing for high-volume XML message brokering. In *VLDB*, 2003.
[16] Y. Diao, S. Rizvi, and M. J. Franklin. Towards an internet-scale XML dissemination service. In *VLDB*, 2004.
[17] Huebsch et al. Querying the internet with PIER. In *VLDB*, 2003.
[18] Kostic et al. Bullet: High bandwidth data dissemination using an overlay mesh. In *SOSP*, 2003.
[19] Kuntschke et al. StreamGlobe: Processing and sharing data streams in grib-based P2P infrastructures. In *VLDB*, 2005.
[20] Liu et al. Client behavior and feed characteristics of rss, a publish-subscribe system for web micronews. In *IMC*, 2005.
[21] Motwani et al. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.
[22] O. Papaemmanouil and U. Çetintemel. Semcast: Semantic multicast for content-based data dissemination. In *ICDE*, 2005.
[23] Papaemmanouil et al. Extensible optimization in overlay dissemination trees. In *SIGMOD*, 2006.
[24] Pietzuch et al. Network-aware operator placement for stream-processing systems. In *ICDE*, 2006.
[25] Srivastava et al. Operator placement for in-network stream query processing. In *PODS*, 2005.
[26] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over stream. In *VLDB*, 2006.
[27] T. Yan and H. Garcia-Molina. The SIFT information dissemination system. In *ACM TODS*, 1999.
[28] Zhou et al. Leveraging distributed pub/sub systems for scalable stream query processing. In *BIRTE*, 2006.
[29] Zhu et al. Dynamic plan migration for continuous queries over data streams. In *SIGMOD*, 2004.