Extensible Profile-driven Data Dissemination

Olga Papaemmanouil, Yanif Ahmad,
Ugur Cetintemel, John Jannotti, Yenel Yildirim

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-06-05
March 2006

Extensible Profile-driven Data Dissemination

Olga Papaemmanouil, Yanif Ahmad, Uğur Çetintemel, John Jannotti, Yenel Yildirim

Department of Computer Science
Brown University
{olga, yna, ugur, jj, yenely}@cs.brown.edu

ABSTRACT

In this paper we introduce XPORT (eXtensible Profile-driven Overlay Routing Trees), a profile-driven distributed data dissemination system that supports an extensible set of data types, profiles, and optimization metrics. XPORT efficiently implements a generic tree-based overlay network, which can be customized per application using a small number of methods that encapsulate application-specific data-profile matching, aggregation, and optimization logic. The clean separation between the "plumbing" and "application" enables XPORT to uniformly and easily support disparate dissemination-based applications such as content-based data feed dissemination and multicast-based content distribution.

We first present the basic XPORT model and its built-in methods. We then describe in detail XPORT's extensible optimization framework that facilitates easy specification of a wide range of useful performance goals. XPORT uses a continuous, adaptive optimization model to achieve these goals under changing network and application conditions. We finally demonstrate the flexibility and the effectiveness of XPORT through real-world data and results from both prototype-based LAN emulation and deployment on PlanetLab.

1. INTRODUCTION

XPORT is a generic profile-driven distributed data dissemination system. Extensibility is the central design consideration for XPORT, which supports an extensible set of data types, profiles, and optimization metrics. XPORT is designed to provide the core data dissemination infrastructure for a growing set of dissemination-based applications and services, including web feed dissemination (RSS/Atom), multicast-based content distribution, large-scale collaborative applications, massively multiplayer network games, and stock ticker distribution.

XPORT is motivated by the observation that even though many dissemination-based applications exhibit diverse application logic and performance requirements, they all require common basic facilities. These include dissemination overlay construction and maintenance, (content-based) routing logic, and membership management. Currently, these applications are often developed from scratch, requiring substantial effort and investment to "get it right" for each specific case. XPORT addresses this problem by providing the core application agnostic facilities, and letting the developers customize and extend their system for a specific target domain. This customization is facilitated through a small number of methods that encapsulate application-specific behavior and optimizations.

XPORT supports two types of extensibility. *Profile-related extensibility* refers to the ability to easily define new data and profile types and is key to supporting a variety of applications. *Cost-related extensibility* refers to the ability to express application-specific performance goals, allowing applications to define their own criterion of an efficient and effective dissemination network. Moreover, users have the ability to extend certain optimization rules. Given user-defined data and profile types, performance metrics and optimization rules, XPORT automatically builds and maintains an overlay dissemination tree consisting of the available broker machines in the system. The system is continuously reconfigured through local transformations, adapting to changes in the network and workload conditions, and incrementally converges to an overlay configuration that is "optimal" (as defined by the application).

The main focus of this paper is on the design, implementation, and evaluation of XPORT's basic optimization framework, which uses a novel two-level aggregation model to define the system's cost. The first level computes the cost of each node as an aggregation of an application-defined cost metric computed over the node's local neighborhood. The second level computes the system cost by aggregating the node costs. This model allows the uniform specification of several useful, commonly used cost functions, as well as new ones, through various combinations of different aggregations functions and local cost metrics. During run time, the system uses a continuous optimization model, where it iteratively applies local transformations to converge to a minimal cost configuration, optionally subject to constraints (specified similarly). XPORT provides a set of built-in base transformation rules, which can be extended by composition.

The key contributions of the paper can be summarized as follows:

1. We introduce the basic design, architecture, and API of XPORT, which, to the best of our knowledge, is the first dissemination system that provides data-profile and cost extensibility.
2. We present in detail XPORT's core optimization framework that uses a metric-independent protocol to optimize user-defined cost functions (subject to constraints). The framework includes a grammar that allows the specification of a large set of cost functions and constraints, as well as an extensible set of transformation rules. We also describe the operational details of an adaptive iterative optimization protocol that implements the framework.

SIGMOD 2006, June 27–29, 2006, Chicago, Illinois, USA.

3. We present experimental evidence, based on real-world data (RSS feeds) and prototype-based results from both LAN emulation and deployment on PlanetLab, that demonstrate the flexibility, practicality, and effectiveness of XPORT.

We begin by introducing the system's API in Section 2. We introduce our optimization framework and provide a detailed discussion of its non-operational aspects in Section 3. We follow up in Section 4 by describing the basic XPORT architecture and run-time system behavior. We describe our experiments and present our results in Section 5, describe related work in Section 6, and conclude the paper with final remarks and our plans for future work in Section 7.

2. XPORT API

In the first part of this section we motivate XPORT's basic API, by discussing the common characteristics of dissemination-based applications. We then describe the methods an application needs to define to express its "native" data and profile types and performance targets.

2.1 Background: Dissemination-based Systems

To introduce XPORT, it is helpful to review profile-driven data dissemination systems in simple terms. The goal here is to highlight the common functionality in these systems and, using this commonality, motivate the general methods used by XPORT.

Profile-driven data dissemination applications typically adopt a declarative, publish-subscribe API that decouples data producers (*sources*) and consumers (*clients*) and isolates both parties from the details of the underlying implementation. The key abstraction is that data producers generate data by *publishing* and data consumers *subscribe* to data by specifying their profiles. The underlying dissemination system is responsible for delivering to each subscriber the data that matches her profile.

The dissemination infrastructure consists of a set of nodes (often called *brokers*) organized into an overlay network. This network usually consists of one or more data dissemination trees [3, 8, 13]. Clients register their profiles with brokers. Profiles are propagated upstream to the root of the tree, creating a reverse routing path. Optionally, profiles are *merged* when possible to reduce routing state requirements and filtering costs.

Using the routing tree, a broker can now send a message only to the subset of its children that is interested in receiving the message, instead of forwarding the message to all its children, thereby eliminating the "flooding" problem. This routing scheme works by *matching* each message with the routing table entries that represent the aggregated profile for each subtree.

Depending on the data types and the complexity of the profiles demanded by the application, data dissemination systems may choose to use their own algorithms and *indexing structures* for efficiently storing profiles on every broker and matching incoming messages against them. For example, ONYX [8] uses YFilter [7] for matching, whereas SIENA [3, 4] uses a custom index [4] for storing and matching relational profiles.

Different dissemination-based systems and applications can have widely varying efficiency targets and constraints. Various latency-related metrics (*e.g.*, matching times, forwarding costs), bandwidth-efficiency metrics (*e.g.*, per-node bandwidth consumption), fairness metrics (*e.g.*, uniform bandwidth utilization across nodes), reliability metrics (*e.g.*, message loss rates), data quality metrics (*e.g.*, fidelity), as well as composite metrics (*e.g.*, product of bandwidth and latency) have been used and studied. Moreover, many systems have commonly limited certain metrics to maintain quality of service (*e.g.*, a maximum end-to-end latency constraint) or control resource usage (*e.g.*, a maximum bandwidth consumption constraint).

```
<DEFAULT METRICS>: path latency | incoming data rate | ...
<DEFAULT STATE>: profile set | merged profile | merged client profile | ...

<LOCAL VALUE>:f(<DEFAULT STATE>|<DEFAULT METRICS>, link | node, <NODE COST SET>) |
                f( link | node, <NODE COST SET>)|
                <DEFAULT STATE>|<DEFAULT METRICS>, <NODE COST SET>

<NODE COST SET>: path | children
<NODE COST FUNCTION>: min | max | sum | product
<NODE COST>: aggregate(<NODE COST FUNCTION>,<LOCAL VALUE>,
                        <NODE COST SET>)|
              (<LOCAL VALUE>,{<NODE COST SET>})

<SYSTEM COST SET>: brokers | clients | brokers−clients
<SYSTEM COST FUNCTION>: min | max | sum | average
<SYSTEM COST>: aggregate(<SYSTEM COST FUNCTION>, <NODE COST>,
                          <SYSTEM COST SET>)

<CONSTRAINT METRIC>: <SYSTEM COST> |<NODE COST>
<CONSTRAINT>: <CONSTRAINT METRIC> <OPERATOR> threshold
<OPERATOR>: < | > | <= | >= | !=
```

Figure 1: XPORT's performance metric grammar.

2.2 Application-defined methods

Based on the main functionality of data dissemination applications, we identified two types of methods an application needs to define in XPORT, *profile-related* and *cost-related* methods. For simplicity of exposition, we abstractly describe these methods, without providing their full signatures or semantics.

2.2.1 Profile-related methods

Profile methods describe how the matching of messages and profiles will be performed. Optionally, the application can also specify how profiles should be locally stored, indexed, and maintained at each node. By default, XPORT stores every new profile received as a separate entry, and uses a simple disjunction operator for profile merging.

- **merge(**p**,** q**)**: Given two profiles p and q, it returns a more general query w covering the two profiles. This function could be used for merging profiles received from clients or child brokers, reducing the routing entries maintained in every node as well as the matching costs.
- **Index-related methods**: XPORT allows applications to integrate an index structure by specifying the following methods:
  - **init()**: declares and initializes the index structure
  - **add(**p**)**: adds a new profile to the index
  - **remove(**p**)**: removes a profile from the index
- **match(**m**,** p**)**: Given a message m and a profile expression p returns true if m matches p or false otherwise.
- **match(**m**,** ind**)**: Given a message m and an index ind used for storing profiles, it returns the set of profiles that match the message.

2.2.2 Cost-related methods

XPORT allows applications to specify their own performance criteria for the dissemination network created. It uses a *two-level aggregation model* to specify system cost. The first level computes the cost of each node as an aggregation of an application-defined cost metric. The second level computes the cost of the whole system by aggregating the node costs. Both aggregations are defined by the general signature:

aggregate(aggregation function, value, aggregation set)

Similarly, applications can also define constraints for each node. Constraints are defined in the same way as the system's cost and additional a threshold can be specified for the constrained metric. Figure 1 shows the XPORT's grammar for defining the performance criteria. In the rest of the section we describe in more detail this grammar and the cost-related methods.

At this point, we note that XPORT nodes maintain some built-in performance metrics like path latency, incoming data rate, etc. Moreover, they maintain some profile-related state, *e.g.* the node's profile, its client profiles, its children aggregated profiles. These are denoted in the grammar by the term `DEFAULT METRICS` and `DEFAULT STATE`, respectively.

- **Local value**: Initially, the application defines the node's local performance metric, which we call *local value*. Applications can either pick a built-in metric, or they can provide a method for computing this metric. As shown in Figure 1, this method can also have as input some of the predefined state variables or metrics. An example local value is the expected incoming data rate, which could be defined as a function of the selectivities of the node's profiles.

  The local value can be a metric referring either to the node itself (*e.g.* CPU usage) or to the links of the node with its neighbors (*e.g.* latency to the parent). This option is specified by the parameter *node* or *link* in the metric implementation function. The exact link on which the local value is determined by the `NODE COST SET` term of the grammar. If this term is set to *path*, then the local value is measured on the link to the node's parent, and if it is set to *children*, it is measured on the links to the children. We note that our definition for the local value allows applications to specify a local value as the combination of optimization metrics, *e.g.*, product of latency and bandwidth. Such combinations customize XPORT to create a multi-metric overlay network.

- **Node cost** : The next metric to define is the node's cost. The node cost could be a) the local value defined previously, b) a built-in state variable, or c) an aggregation of the local values of some neighboring nodes. This aggregation function is defined as:

  aggregate(node cost function, local value, node cost set)

  The *node cost function* can be SUM, PRODUCT, MIN and MAX. There are two options in terms of the *node cost set*; it could be either the nodes on the path to the tree's root (referred as *aggregation over path*), or the node's immediate children in the tree (referred as *aggregation over children*). The above method allows applications to define a large set of metrics, used frequently for the evaluation of dissemination based systems. An example metric defined as aggregation over the path is the path latency; this is the sum of the latency of every link on the path to the root. Outgoing bandwidth consumption is a metric defined as an aggregation over the children; it is the sum of the incoming data to each child.

- **System cost**: Finally, the application defines the system cost. This is an aggregation of the node cost over all the nodes in the system:

  aggregate(system cost function, node cost, system cost set)

  We distinguish three categories for the system cost function: a) *additive functions* (SUM)[1], b) *bottleneck functions* (MIN, MAX) and c) *statistical functions* (VARIANCE).[2] The aggregation set, *system cost set*, allows the system cost to be aggregated over the broker or client set. The definition for the system cost permits applications to define a large variety of system cost measures, like maximum latency, total bandwidth consumption, or average matching overhead. An example metric is the average path latency. In this case each node's cost is the path latency and XPORT's cost is defined as the average over all nodes in the dissemination tree.

- **Constraints**: Constraints can be defined as:

  constraint(metric, operator, threshold)

  This method allows a threshold to be specified for every node. The definition of the constraint metric follows the 2-level aggregation model, and it can be specified in the same way as the node cost or the system cost. For example, an application might want to impose an upper bound on the path latency of every node, which is one-level aggregation of the link latency over the path to the root, as we mentioned above. Similarly, a dissemination system might try to guarantee a lower bound of the maximum path latency. This a two-level aggregation on the path latency of every node in the system. XPORT customizes its functionality and optimization framework to respect these constraints.

Aggregation Combinations	System Cost Function	Node Cost Function	Example Metrics
Type I	sum	sum /product	total (average) path latency
Type II	sum	min / max	total path bandwidth bottleneck
Type III	min/max	sum / product	maximum path reliability
Type IV	min /max	min / max	min path bandwidth bottleneck
Type V	variance	sum/product	variance of path latency
Type VI	variance	min/max	variance of bandwidth bottleneck

Table 1: Combinations of aggregation functions

We categorize the combinations of aggregation functions used for node and system cost calculation into six types. These types, along with some example metrics, are shown in Table 1.

2.2.3 Cost metric examples

In this paragraph we provide some example metrics. In the first example, we calculate the *average path latency*. Every node measures the link latency to its parent and adds this to the path latency of its parent:

`system cost`	=	*aggregate*(AVG, *path latency*,`brokers`)
path latency	=	*aggregate*(SUM, *link latency*, `path`)

If the system cost is the *bandwidth bottleneck* of the system, every broker measures the bandwidth bottleneck of its path to the root (*i.e.*, the link with the minimum bandwidth capacity) and the cost definition is:

`system cost`	=	*aggregate*(MIN, *bandwidth*, `brokers`)
bandwidth	=	*aggregate*(MIN, *link bandwidth*, `path`)

An example of a metric with no aggregation for the node cost is the *total superfluous incoming data*. In this case the application wants to minimize the undesired data each broker receives and processes. This is the data the node is not interested in receiving, and simply has to forward it to its descendants.

`system cost`	=	*aggregate*(SUM, *superfluous data*, `brokers`)
superfluous data	=	(`extraIncoming()`, `parent`)

[1]Average can also be expressed as the sum divided by the size of the aggregation set

[2]Standard deviation can also be expressed as the square root of variance.

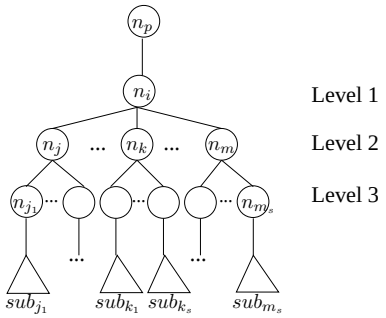

Figure 2: 3-level optimization unit for n_i. Only nodes in levels 1, 2 and 3 belong to the unit.

```
//merged_client_p is the merged profile of the
  clients connected on that node
//merged_p is the merged profile of the
  client and child profiles
extraIncoming( merged_client_p, merged_p,link ){
  double total_in=matching_rate(merge_p);
  double desired_in=matching_rate(merged_client_p);
  return (total_in - desired_in);
}
```

3. OPTIMIZATION FRAMEWORK

XPORT strives to create dissemination trees that benefit application specific performance metrics. Periodically, it modifies the overlay tree using local transformations to adapt to changing network conditions or data changes that might affect the system's performance. We define a *local transformation* as one that requires interactions among "nearby" brokers on the overlay tree. In our current implementation, these brokers are at most three levels from each other. These include a broker n_i, its parent n_p, its children and grandchildren, as shown in Figure 2, and we call them the *optimization unit* of broker n_i, $unit_i$.

Our local transformations do not affect the optimization unit's interface with the rest of the network[3]. This implies that the path from n_p to the root and the subtrees below the third level of the unit in Figure 2 will not be affected, in terms of their topology and profile. Thus, both the parent of n_p and the brokers in the last level will continue forwarding the same data to the same brokers, as before the transformation. The only brokers that might experience a change in their connections and profile are those in the optimization unit. However, any node's cost might be affected by a transformation. Keeping the topological and profile-related effects of our optimizations local (*i.e.*, only to the nodes in the optimization unit) allows us to maintain information only for the nodes inside the unit and reduces the cost of network reconfiguration, as fewer nodes are affected by each transformation.

3.1 Local transformations

A greedy approach for identifying the best transformation of an optimization unit would be to consider all the possible reconfiguration of this unit's structure. There are two main drawbacks of this approach. The first one is the exponential size of configurations that have to be considered. The second drawback is the increased communication overhead; the information exchanged between nodes trying to quantify the benefit of each transformation would be significantly high. XPORT's optimization framework uses a set of primitive transformations instead of performing an exhausting search of all possible configurations. Moreover, it allows applications to extend this set by defining their own transformations. As our experimental results show, even a small number of transformation could lead to significant performance improvements.

[3]We note that our transformations do not migrate client connections.

Transformation	Definition
Demote child	$demote(n_k, n_j)$
Promote child	$promote(n_j)$
Subtree promotion(n_j)	$demote(n_{j_s})^*$
Parent-child swap(n_j)	$promote(n_j) \rightarrow$ $demote(n_i, n_j) \rightarrow$ $promote(n_{i_s})^* \rightarrow$ $demote(n_{j_s}, n_i)^*$
Subtree migration(n_k, n_j)	$promote(n_{j_s})^* \rightarrow$ $demote(n_{j_s}, n_k)^*$
Sibling swap(n_j, n_k)	$promote(n_{j_s})^* \rightarrow$ $demote(n_{j_s}, n_k)^* \rightarrow$ $promote(n_{k_s})^* \rightarrow$ $demote(n_{k_s}, n_j)^*$

Table 2: Primitive and composite transformations.

XPORT's *primitive transformations* are *child demotion* and *child promotion*, shown in Figure 3 (a) and 3(b) respectively. We explain these transformations with respect to the 3-level optimization unit shown in Figure 2:

- **Demote child.** This transformation picks a node n_k from the second level of the unit, and moves it along with its subtree, under one of its siblings n_j. This increases the number of subtrees of n_j, leaving n_i with one less subtree.

- **Promote child.** This transformation picks a node n_{j_1} from the third level in the optimization unit and moves it along with its subtree sub_{j_1} under its grandparent n_i. This increases the number of subtrees of n_i, leaving n_j with one less subtree.

3.1.1 Extending the transformation set

XPORT's transformation set is extensible. Applications can define their own *composite* transformations by using the primitive ones. Allowing composite transformations is important for our system performance as they help avoid being stuck in a local optimal. Thus, they can help the system improve its performance faster, than if we were using only the two primitive ones. In our prototype, we defined the following four composite transformations:

- **Subtree promotion.** In this operation the subtree sub_j of a node n_j in the second level of the optimization unit is moved under its parent n_i. This will increase the children of n_i, leaving broker n_j with an empty subtree (shown in Figure 3 (c)). This transformation can be derived by applying the promote child operation for every child n_{j_s} of node n_j.

- **Parent-child swap.** As Figure 3 (d) shows, in this transformation the root of the optimization unit n_i and its child n_j swap positions, without moving the subtree of n_j. This will force every child and subtree previously under n_i to have a different parent. Parent-child swap can be derived from combining the promote and demote child.

- **Subtree migration.** In this case the subtree of a node n_j in the second level of the optimization unit, migrates under its sibling node n_k. Node n_k remains with no children, while n_j's subtree increases by the subtree of n_k. Subtree migration can be derived from the two primitive transformation, by first promoting all the children of n_j under n_i and then demoting the same nodes under n_k. This transformation is shown in Figure 3 (e).

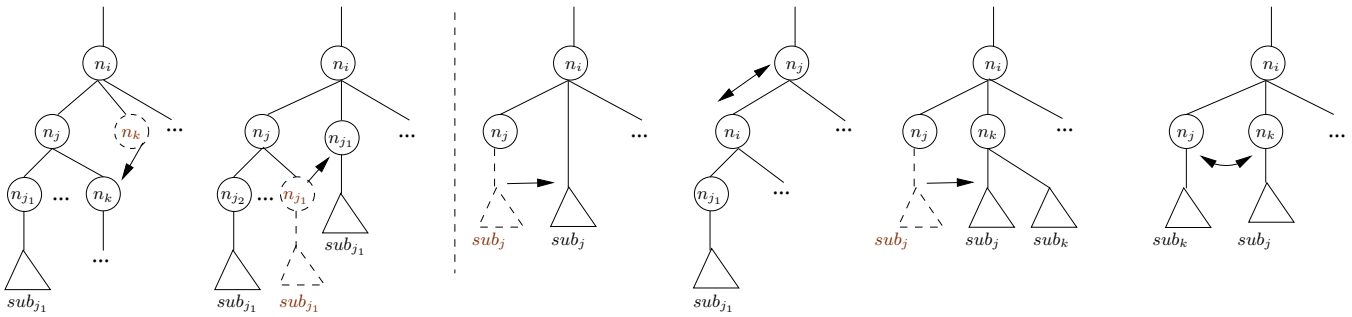

Figure 3: XPORT's transformations. (a) and (b) are the primitive transformations. (c), (d), (e), and (f) are composite transformations specified as a sequence of primitive transformations.

- **Sibling swap.** In this transformation two siblings n_k and n_j swap positions. This will change the root node of the subtrees below these nodes, as shown in Figure 3 (f). Sibling swap can be expressed as promoting the children of n_j under n_i, and then demoting them under n_k and performing the same for the children of n_k.

Table 2 shows how these transformations are created from the primitive ones[4].

Extending the scope of the optimization unit. As mentioned earlier, XPORT's optimization unit is limited to nodes within three levels of each other. However, the optimization unit can be extended to multiple levels. Extending the scope of the unit, increases the flexibility of the system, as it allows the application to extended even further the transformation set. Although, such an extension increases the opportunities for optimization, also increases the maintenance traffic and the state nodes need to maintain.

3.2 Quantifying a transformation's benefit

The goal of the local transformations is to improve the overall system cost. Thus, XPORT calculates the cost benefit of every transformation and applies the best one. Since the system cost is an aggregation over the node costs, the exhaustive approach for quantifying the cost improvement of a transformation is to estimate the cost of every node after the transformation and aggregate them to get the new system cost. This approach has high communication cost, but since XPORT understands the semantics of the aggregation functions, it can automatically quantify the effects of a transformation. Moreover, XPORT automatically identifies the state a node needs to maintain in order to quantify the benefit of a transformation, and also the information to be exchanged among nodes during the optimization. In this section we will describe this approach. In the first part we provide the general equations that estimate the cost benefit of every transformation. In the second part we describe how we quantify efficiently, and with small communication cost, this benefit.

DEFINITION 1. *Let $cost_i$ denote the cost of node n_i. The* dependence set *D_i of n_i is the set of nodes whose cost is affected by a change in $cost_i$. In particular, D_i includes the nodes of the subtree defined by n_i (resp. the nodes of the path from n_i to the root) when $cost_i$ is calculated as aggregation over the path (resp. aggregation over the children). We refer to members of D_i as the* dependents *of n_i.*

For example, for aggregation over path, a change of n_i's latency to its parent, will affect the path latency of every node in its subtree. Similarly, for aggregation over children, a change on a child's profile could affect the outgoing bandwidth consumption of the nodes on its path to the root, as they may need to forward downstream a different data set.[5]

However, if XPORT uses only one level of aggregation, *i.e.*, there is no aggregation for calculating the node cost, the dependence set of a node may be all the nodes in the network. XPORT can limit this set based on the definition of the local value used as the node cost. For example, if the node cost is defined as a function of the merged profile of n_i, then since a transformation could affect only the profiles of the nodes in its path to the root, then n_i's dependents are the nodes in this path.

DEFINITION 2. *The* dependence set cost *$cost(D_i)$ of n_i, is the aggregation of $cost_j, n_j \in D_i$, over D_i, using the system cost aggregation function. Then a change of $cost_i$ may affect $cost(D_i)$. We denote this change as $\Delta cost(D_i)$ and the new dependence set cost as $cost'(D_i)$.*

For example, if the system cost aggregation function is MIN, then

$$cost(D_i) = \min_{j \in D_i} \{cost_j\}$$

In the following sections, we assume that a transformation is performed in n_i's optimization unit. Let U_i denote the set of nodes in n_i's optimization unit, and S_i the dependence sets of the nodes in the last level of n_i optimization unit. Thus, for Figure 2, we have:

$$U_i = \{n_p, n_i, n_j, ..., n_k, ..., n_m, n_{j_1}, ..., n_{k_1}, ..., n_{k_s}, ..., n_{m_s}\}$$

$$S_i = \{D_{j_1}, ..., D_{k_1}, ..., D_{k_s}, ..., D_{m_s}\}$$

Finally, we denote $\Delta cost_i$ as n_i's cost change after the transformation, $cost'_i$ as its new cost value, and b_i as the benefit of the transformation with respect to the system cost.

We provide now the general formulas, that quantify the cost benefit of a transformation in n_i's optimization unit. These formulas can be separated into two terms, one that calculates the cost change of the nodes inside the optimization unit (*i.e.* $n_j \in U_i$) and one that

[4]In Table 2 '*' implies that the operation will be performed repeatedly for every node at the same level and with same parent as the first parameter of the transformation.

[5]If change in $cost_i$ results from a transformation, then D_i can be restricted to the path to the root of the optimization unit the transformation is performed on. This is due to the fact that transformations maintain the same interface with the rest of the network, thus nodes higher than the unit will not be affected.

calculates the cost change of the nodes outside the unit that can be affected by its transformation (*i.e.* $n_j \in S_i$)

Additive functions. In this case, the system cost is the sum of the node costs. Thus, b_i is the sum of the cost change of all nodes affected by the transformation:

$$\begin{aligned} b_i =& (cost'_i - cost_i) + \sum_{j \in U_i} (cost'_j - cost_j) + \\ &+ \sum_{k \in D_m | D_m \in S_i} \Delta cost(D_k) \end{aligned} \quad (1)$$

Formula 1 is the general formula that holds for every transformation. However, depending on the transformation, many of its terms are zero, so simpler formulas can be used. Examples of these formulas are given in Table 3. For example, in the child promotion operation, only the child to be promoted and the nodes in its subtree are affected, so only these nodes can change the system cost.

Bottleneck functions. For the bottleneck functions, n_i estimates the new system cost and compares it with the current system cost. Thus, if c denotes the current system cost and the system cost function is MIN, then:

$$\begin{aligned} b_i &= c - \min_{j \in U_i, k \in S_i} \{cost'_i, cost'_j, cost'(D_k)\} \\ &= c - \min_{j \in U_i, k \in D_m | D_m \in S_i} \{cost'_i, cost'_j, \\ & cost(D_k) + \Delta cost(D_k)\} \end{aligned} \quad (2)$$

Statistical functions. In this case, n_i calculates the new cost of every node in the system, and estimates the new variance value. Thus:

$$\begin{aligned} b_i &= c - \{\frac{1}{|V|-1} \sum_{j \in V} (cost'_j - \overline{cost'})^2\} \\ &= c - \{\frac{1}{|V|-1} \{\sum_{j \in U_i} (cost'_j - \overline{cost'})^2 + \\ & \sum_{j \in D_m | D_m \in S_i} ((cost_j + \Delta cost_j) - \overline{cost'})^2 + \\ & \sum_{j \notin U_i, D_m | D_m \in S_i} (cost_j - \overline{cost'})^2\}\} \end{aligned} \quad (3)$$

where $\overline{cost'}$ is the average node cost after the transformation.

Extending the optimization framework. We mentioned earlier that the scope of the optimization unit and the transformation set of XPORT can be extended. Here, we describe the impact of these extensions to the way XPORT quantifies the benefit of every transformation.

Increasing the size of the optimization unit, will simply increase the number of terms in Formulas 1, 2 and 3, adapting them to work for the new optimization unit.

Defining new transformations is simple from the user perspective, she has to define which transformations will be combined along with the desired parameters. Defining a new transformation, from the system perspective, requires the definition of the equation that quantifies its cost benefit. This formula can be derived by using Formulas 1, 2 or 3 and aggregating them over all the transformations using the system cost aggregation function. For example, it is straightforward to see that, for the subtree promotion, the equation is simply a sum of the formula of the promote child operation, over all children of node n_j. Moreover, for more complicated composite transformations, like sibling swap, more compact formulas will be derived, as many of their terms are cancelled out. Providing compact formulas implies less traffic during optimization, due to the smaller state exchanged among nodes in the optimization unit.

3.2.1 *Quantifying cost changes outside n_i's optimization unit*

The formulas that quantify the benefit of a transformation, require estimating the cost effect on nodes outside the optimization unit. For the additive and bottleneck function, the formulas require the change on the dependence set cost of some nodes, and for the statistical functions they require cost change of the nodes outside the unit.

XPORT strives to incur the minimum communication overhead, when estimating these cost effects. Therefore, it tries to avoid any metadata exchange with nodes outside an optimization unit. Instead, in order to be able to calculate the effect of the transformation on these nodes, nodes maintain some metadata regarding nodes outside the optimization unit they own. We refer to this state as the *transformation state*.

XPORT exploits the characteristics of the aggregation functions to identify both the minimum transformation state required and to derive general formulas that quantify the cost change of a node outside an optimization unit. We provide the details of our approach in the following paragraphs, for the different types of node cost aggregation.[6]

3.2.1.1 *Node cost function: SUM/PRODUCT.*

For the purpose of illustration, we focus on the case of the SUM function and an aggregation over the path to the root. Our results are similar for the case of a PRODUCT function and an aggregation over the children set. An example metric is the path latency.

In this case, a change of a node's cost (*i.e.*, path latency to the root) will change the cost of every dependent (*i.e.*, every node in its subtree) by the same amount. Given this, we describe the formulas for quantifying the cost change of a dependence set, for the different types of system cost functions.

Additive functions. Since the system cost is defined as the sum of all node costs, we are interested in the total cost change over all the dependents, which is :

$$\Delta cost(D_i) = \Delta cost_i \times |D_i|$$

Bottleneck functions. Here, the system aggregation function is either MIN (or MAX). Thus, we are only interested in the change on the minimum (or maximum) node cost among the dependents. This change is given by:

$$\Delta cost(D_i) = \Delta cost_i$$

Statistical functions. Here, the cost of every dependent changes by

$$\Delta cost_j = \Delta cost_i$$

3.2.1.2 *Node cost function: MIN/MAX.*

We focus here on the MIN aggregation function, for the purpose of illustrating our main ideas. Our results can be easily extended for the MAX function. Again, we assume that an aggregation over the path to the root, and for simplicity we assume that the local value of a node is measured over the link to its parent. An example metric is the bandwidth bottleneck of a node, *i.e.*, the bandwidth capacity of its path to the root. This is the minimum bandwidth capacity over all the links on that path. Here on, we will refer to this bandwidth capacity as the bottleneck value, and the link (node) with this bandwidth capacity as the bottleneck link (node). For simplicity, we will present our solution with respect to this metric.

[6]When no aggregation is used for the node cost definition, XPORT uses Formulas 1, 2 and 3 to quantify the result of a transformation by recalculating the cost of every node outside the unit.

demote child	$(cost'_k - cost_k) + \Delta cost(D_k)$
promote child	$(cost'_j - cost_j) + \Delta cost(D_j)$
subtree promotion	$\sum_{k \in children_j} \Delta cost(D_k) + \sum_{k \in children_j} (cost'_k - cost_k)$
subtree migration	$\sum_{j \in children_k} \Delta cost(D_j) + \sum_{j \in children_k} (cost'_j - cost_j)$
sibling swap	$\sum_{s \in children_k, chilren_k} \Delta cos(D_s) + \sum_{s \in children_k, children_j} (cost'_s - cost_s)$

Table 3: Cost formulas for local transformations (when system cost aggregation function is SUM).

In this case, a change in the bandwidth capacity of a link, may affect the bandwidth bottleneck of the downstream nodes that have this link as their bottleneck link. Consequently, a change in a node's n_i cost by $\Delta cost_i$, may affect the cost of every dependent n_j(*i.e.*, a node in its subtree). Obviously, this effect depends on the links lying between the node n_i and n_j. The metric we define below, allows to detect these links.

DEFINITION 3. *The* cost of n_i's dependent n_j relative to n_i, $h_i(j)$, *is the aggregation of the local values of all nodes lying on the path connecting n_i and n_j, in the dissemination tree, using the node cost aggregation function. The aggregation of n_i's local value $local\_value_i$ and $h_i(j), \forall n_j \in D_i$, is referred to as the* minimum local value of n_i, g_i.

In the case of the bandwidth bottleneck metric, $h_i(j)$ identifies the minimum bandwidth capacity link, between n_i and a node n_j in its subtree, while g_i identifies the minimum bandwidth capacity of the links in n_i's subtree.

Each node n_i, maintains the above metric, for every one of its dependents. In order to reduce the amount of state, a node n_i stores only the unique $h_i(j)$ values along with the frequency for each distinct value:

$$T_i = \{(\lambda, \alpha)\}, \text{ where } \lambda = |\{n_j | n_j \in D_i, h_i(j) = \alpha\}|.$$

This is possible in the case of the additive and bottleneck system aggregation functions.

Additive functions. In this case, we are interested in the total cost change over all the dependents. We distinguish two cases; one when the cost of n_i decreases and one when it increases. For the first case, the cost change is:

$$\Delta cost(D_i) = \begin{cases} \Delta cost_i \times |D_i| & g_i \geq cost_i \\ \gamma & \text{otherwise} \end{cases}$$

where

$$\gamma = \sum_{\alpha} (\min\{cost_i, \alpha\} - cost'_i) \times \lambda, \text{ where } (\lambda, \alpha) \in T_i, \alpha \geq cost'_i$$[7]

In the first case, n_i and its dependents share the same bottleneck link, so their cost is affected by the same amount $\Delta cost_i$. The second option refers to the case, where some dependents with a different bottleneck value before the transformation, share the same bottleneck link with n_i after the transformation. The term γ calculate their total cost change.

In the second case, where the cost of n_i increases, the total cost change is:

$$\Delta cost(D_i) = \begin{cases} \Delta cost_i \times |D_i| & g_i > cost_i, \text{ and } g_i > cost'_i \\ \omega & g_i > cost_i \text{ and } g_i \leq cost'_i \text{ or} \\ & g_i \leq cost_i \end{cases}$$

where

$$\omega = \sum_{\alpha} \Delta cost_i \times \lambda + \sum_{\alpha'} (\alpha' - cost_i) \times \lambda',$$

where $(\lambda, \alpha) \in T_i, \alpha \geq cost'_i$ and $(\lambda', \alpha') \in T_i, \alpha' < cost'_i$

The term ω calculates the total cost change of the dependents that experience the same cost change as n_i, and the cost change of the nodes that have a lower bottleneck link after the transformation. **Bottleneck functions.** In this case we are interested in the minimum change among all the dependents. We distinguish again two cases; one when the cost of node n_i decreases and one when it increases. For the first case:

$$\Delta cost(D_i) \begin{cases} \Delta cost_i & g_i \geq cost_i \\ g_i - cost'_i & g_i > cost'_i \\ 0 & g_i \leq cost'_i \end{cases}$$

The first term refers to the case where the bottleneck value of the dependents do not change, the second to the case when a dependent has a different bottleneck value after the transformation, and the third one to the case where n_i and its dependent share the same bottleneck link.

Similarly if $cost_i$ increases then:

$$\Delta cost(D_i) = \begin{cases} 0 & g_i \leq cost_i \\ \Delta cost_i & g_i > cost_i \text{ and } g_i > cost'_i \\ cost'_i - g_i & g_i > cost_i \text{ and } g_i \leq cost'_i \end{cases}$$

Statistical functions. We distinguish two cases; one when the cos_i decreases and one when it increases. For the first case, the cost change is:

$$\Delta cost_j = \begin{cases} \Delta cost_i & g_i \geq cost_i \\ (\min\{cost_i, h_i(j)\} - cost'_i) & h_i(j) \leq cost'_i, g_i \leq cost'_i \text{ or} \\ & h_i(j) \leq cost'_i, g_i > cost'_i \end{cases}$$

In the first case, n_i and the dependent n_j share the same bottleneck link, so their cost is affected by the same amount $\Delta cost_i$. The second option refers to the case, where the dependent had a different bottleneck value before the transformation, but shares the same bottleneck link with n_i after the transformation.

In the second case, where $cost_i$ increases, the total cost change is:

$$\Delta cost_j = \begin{cases} \Delta cost_i & h_i(j) \geq cost'_i, g_i \leq cost_i \text{ or} \\ & g_i > cost_i, g_i > cost'_i \text{ or} \\ & h_i(j) \geq cost'_i, g_i > cost_i, g_i \leq cost'_i \\ cost'_i - h_i(j) & h_i(j) < cost'_i, g_i > cost_i, g_i \leq cost'_i \end{cases}$$

Aggregation	T	Size	O	Size
Type I	$\|D_i\|$	$O(1)$	$\|D_i\|$	$O(1)$
			$cost_i, cost'_i$	$O(1)$
Type II	$\|D_i\|$	$O(1)$	$\|D_i\|$	$O(1)$
	$h_i(j)$	$O(\|D_i\|)$	$h_i(j)$	$O(\|D_i\|)$
			$cost_i, cost'_i$	$O(1)$
Type III	$cost(D_i)$	$O(1)$	$cost(D_i)$	$O(1)$
	c	$O(1)$	$cost_i, cost'_i$	$O(1)$
Type IV	$cost(D_i)$	$O(1)$	$cost(D_i)$	$O(1)$
	g_i	$O(1)$	g_i	$O(1)$
	c	$O(1)$	$cost_i, cost'_i$	$O(1)$
Type V	$cost_j, j \in S_i$	$O(\|S_i\|)$	$cost_j, j \in S_i$	$O(\|S_i\|)$
	c	$O(1)$	$cost_i, cost'_i$	$O(1)$
Type VI	$cost_j, j \in S_i$	$O(\|S_i\|)$	$cost_j, j \in S_i$	$O(\|S_i\|)$
	$h_i(j)$	$O(\|D_i\|)$	$h_i(j)$	$O(\|D_i\|)$
			$cost_i, cost'_i$	$O(1)$

Table 4: Transformation and optimization state (T refers to transformation state and O to optimization state)

3.2.2 *Statistics approximation*

In the previous section, we showed that maintaining state for nodes outside n_i's optimization unit (*i.e.* dependence set cost, min-

[7] when $\alpha < cost'_i$, then $(\min\{cost_i, \alpha\} - cost'_i) = 0$

imum local value) allows n_i to quantify the effect of a transformation. In order for n_i to calculate this state, its peers need to periodically broadcast some meta-data. This meta-data includes the node's cost and local value. For the case of the additive and bottleneck function, n_i needs to collect this data only from its dependents, while in the case of the statistical function, *i.e.* variance, nodes need to know the cost of every node in the system. To reduce the high overhead of these broadcasts, we vary their frequency.

Thus, XPORT broadcasts more frequently the cost of nodes that have higher impact on the system cost and on their dependents cost. Each node n_i is assigned a weight w_i, depending on this impact. XPORT nodes broadcast their metadata in *broadcast phases*. During each phase, only a subset of the nodes will send its metadata, depending on their weight. Node n_i participates in the broadcast phases with period:

$$w_i \times p, \tag{4}$$

where $p > 1$ is a predefined constant, that allows us to increase the period of each node's broadcast.

The weight of a node depends on the node cost aggregation function. We distinguish two cases; the first one is when this function is SUM or PRODUCT. An example metric, is a node's path latency. In this case, the more dependents a node has, the more nodes it can affect, if its own cost or local value changes (*i.e.* changes on the links closer to the root, affect more nodes, than the links of the leaf nodes). Moreover, having a good estimation of the , allows us to approximate the cost of their dependents (*i.e.* the path latency of a node will be at least as high as the path latency of its parent). Thus, we set the weight n_i to be $w_i = \frac{|V|}{|D_i|}$.

The second case is for the MIN and MAX functions. An example metric, is the bandwidth bottleneck of a node, *i.e.*, the link on its path with the minimum bandwidth capacity. Here, a change on the link of every node on the path, could effect the system cost, as every link could potentially become the bottleneck. However, assuming , nodes with value closer to the bottleneck of their dependents have a higher change of becoming the bottleneck, thus they should broadcast their cost more frequently, than the ones with high difference. Thus, we set the weight n_i to be $w_i = local\_value_i - g_i$.

4. RUN-TIME FUNCTIONALITY

In this section we describe the run time functionality of XPORT. We start the section with the system model and the nodes' architecture and local state. Finally, we describe the control flow of our distributed optimization framework.

4.1 System model and node architecture

System model. XPORT consists of a set of nodes organized into an application-level overlay network. New nodes select an existing node as its parent in the dissemination tree. The parent selection attempts to affect the system cost to the minimum degree, and also to respect the system constraints.

XPORT's model is based on a publish/subscribe paradigm; sources publish their data and clients express their data needs by providing their profiles. Clients connect to an XPORT node and forward their profile to it. Nodes add a new profile p as a new routing entry, or invoke the add(p) function as described in Section 2 if an index is used for storing the profiles. The new profile is merged with the existing ones, using the merge function of XPORT's API. The new profile (or the merged one) is propagated upstream on the path to the source, until a parent with a more general profile (*i.e.*, with no changes in its profile set) or the tree's root node is reached.

Upon receipt of a data message m, a broker checks all the entries in its routing table to determine whether it should be forwarded to a downstream broker or a client. For each entry p in the routing table, the match(m,p) is invoked (or the match(m, ind) if an index ind is used). The output is the set of children or clients with profiles that match m, and m is forwarded to them.

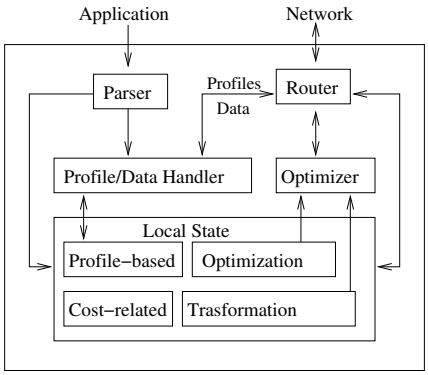

Figure 4: Basic node architecture

Node architecture. The basic architecture of an XPORT node is shown in Figure 4. Application defined methods are given to each node's *parser*, which customizes various system components to implement the application specific profiles, data types and performance metrics. The *data/profile handler* is the component responsible for storing, indexing and maintaining profiles. It also performs the matching of incoming messages against the local profiles. Network transformations are identified and applied by the *optimizer*. All components communicate with the node's *router*, which communicates data and metadata with other brokers. All components have access to the local state of the node, which consists of workload statistics, profile and data information and optimization metadata. In the next paragraph, we describe in more detail this local state.

Local state. The local state includes four types: *profile-based* state, *cost-based* state, *transformation* state, and *optimization* state.

A broker's profile-based state includes the profiles it receives from each of its children brokers and clients, and the merged profile of its children. Thus, the state for n_i is $O(|children_i|)$ in size. The cost-based state includes the local value of the node (if this not one of the profile-based state variables), and the cost of the node. This state is of constant size, $O(1)$.

The transformation state refers to the data that every node has to maintain in order to participate in the local transformations. This state allows a node to quantify the cost benefit of local transformations. Optimization state refers to the information nodes need to exchange with their neighbors during optimization. Optimization and transformation state along with their sizes are given in Table 4.

4.2 Optimization data flow

XPORT employs a *bottleneck-based* approach for optimization. In our approach, the system tries to optimize the system cost function, for the entire system. For example, if the optimization goal is to minimize the maximum CPU load, then the bottleneck approach will attempt to reduce the load of only the most loaded node in the network.

For bottleneck optimization, we rely on the notion of a *critical node*. A node is considered critical if its node cost change may affect the system performance, *e.g.*, if the system cost is the maximum path latency, then the critical nodes are the ones on the path with the maximum latency. Moreover, we maintain the notion of a *critical optimization unit*. We define a critical optimization unit as one that may affect the cost of a critical node, and thus the system

cost. At each *optimization period*, nodes consider local transformations. In the bottleneck approach, only nodes inside a critical optimization unit will attempt to optimize. Note that, our approach guarantees cost improvement in every optimization period, given that at least one beneficial transformation is identified.

During the optimization period, every node owning a critical optimization unit, exchanges data (*i.e.*, optimization state) with the nodes in the unit, and quantifies the benefit of every primitive and composite transformation defined. In every critical unit, the most effective transformation is selected and pushed to the root of the dissemination tree, which identifies and applies the transformation with the maximum benefit. Nodes are informed about the new structure and change their connections to reflect the new tree organization.

During optimization and network reorganization, nodes keep receiving data from the current tree. Once the new connections are set up, they switch to start receiving data from the new tree. To avoid data loss after node disconnections or transformations of the tree, every message pushed to XPORT is assigned a sequence number. Nodes store the sequence number of the last message they have received. Once a node connects to a new parent, either due to a transformation or a lost connection with its old parent, it forwards the latest sequence number it has seen to this parent. This node compares its own latest sequence number and either returns the lost messages, if they are stored in its cache, or forwards the request upstream in the tree, searching for a more updated node. This operation allows our transformations to be transparent to our clients, as no messages will be lost from tree reorganizations.

Opportunistic optimization. A different optimization approach is the *opportunistic* optimization. This approach optimizes the cost function locally for each optimization unit, as opposed to the bottleneck approach, which optimizes the system cost function for the entire system. For example, if the optimization goal is to minimize the maximum CPU load, then the opportunistic approach will reduce the workload of the most loaded node in each optimization unit. Moreover, all units would participate in the optimization period, as opposed to only the critical ones. The main benefit of the opportunistic optimization it that it can indirectly lead the system to a globally near-optimal configuration. However, it cannon guarantee cost improvement in every optimization period. In our current implementation, we used the bottleneck-based approach.

4.2.1 Concurrent transformations

Another optimization issue is the number of transformation units that can be optimized in parallel. The main benefit of a single transformation approach is that it is easier to handle, as we do not need to consider potentially interfering concurrent transformations across multiple optimization units. However, serializing transformations slows down the convergence to the "optimal" topology. We note that, multiple transformations could be performed in a single optimization period as long as they are applied on independent optimization units.

Moreover, XPORT allows concurrent transformation to be applied on non-independent optimization units. In the following paragraphs, we describe our approach. We start with the definitions for independent optimization units and independent transformations.

DEFINITION 4. *Let $unit_i$ and $unit_j$ denote the optimization units of n_i and n_j, respectively. We say that $unit_i$ and $unit_j$ are* independent, *if*

$$(U_i \cup S_i) \cap (U_j \cup S_j) = \oslash.$$

DEFINITION 5. *Let $trans_i$ and $trans_j$ denote transformations of $unit_i$ and $unit_j$, respectively. Let also $trans\_set_i$ and $trans\_set_j$ be the set of nodes affected topologically (i.e., have a different parent or different children set) by $trans_i$ and $trans_j$, respectively. We say that $trans_i$ and $trans_j$ are* independent, *if $unit_i$ and $unit_j$ are independent or*

$$(trans\_set_i \cap trans\_set_j) = \oslash.$$

XPORT allows two transformations to be applied in parallel, *i.e.* in the same optimization period, if they are independent. This ensures that these transformations can both be defined correctly on overlapping optimization units.

Identifying which transformations can be parallelized is the first step for supporting concurrent transformations. The second step is quantifying the cost effect of these transformations when applied in parallel. This cost effect will be compared with the affect of applying only one of the two transformations, in order to decide the best combination.

If the transformations are applied on independent units, then the two transformations can not negatively interfere with each other. Otherwise, one transformation might affect the cost of some nodes in the other unit. Thus, we need to quantify the benefit of parallelizing them. We denote this benefit as b_{ij} and we describe our approach for estimating this benefit in the next paragraphs.

Additive functions. The total benefit of two parallelized transformations is given the following equation:

$$b_{ij} = b_i + b_j$$

Bottleneck functions. For this case we distinguish two cases; the first one is when the $unit_i$ and $unit_j$ are independent. Then:

$$b_{ij} = \min\{b_i, b_j\}.$$

For the second case, $unit_i$ and $unit_j$ are not independent. Thus, assuming that the $unit_i$ includes nodes in higher levels than $unit_j$:

$$\begin{aligned} b_{ij} &= c - \min_{m \in U_j, k \in S_j} \{cost'_j + x, cost'_m + x, cost'(D_k)\} \\ &= c - \min_{m \in U_j, k \in S_j} \{cost'_j + x, cost'_m + x, \\ &\quad cost(D_k) + \Delta cost(D_k) + x\} \end{aligned}$$

where x is the cost increase of the n_k's first ancestor that belongs in $unit_i$ and not in $unit_j$.

Statistical functions. Similarly, if the system cost aggregation function is the variance, then:

$$\begin{aligned} b_i &= c - \{\frac{1}{|V|-1} \sum_{j \in V} (cost'_j - \overline{cost})^2\} \\ &= c - \{\frac{1}{|V|-1} \{ \sum_{m \in U_j} (cost'_m - \overline{cost})^2 + \\ &\quad \sum_{k \in S_j} ((cost_m + \Delta cost_m + x) - \overline{cost})^2 + \\ &\quad \sum_{m \notin S_j, U_j} (cost_m - \overline{cost})^2\}\} \end{aligned}$$

5. PERFORMANCE EVALUATION

We have implemented an XPORT prototype and deployed our implementation across the PlanetLab testbed. We also built an RSS feed disseminates dissemination application using XPORT's API. Using XPORT's API, we built an RSS feed dissemination tree. We implemented an XPORT proxy, which clients can use to instantly allow their existing feed readers to be part of the XPORT dissemination network.

Clients request their desired RSS feeds, by forwarding their profile to an XPORT node. XPORT nodes self-organize in a profile-based dissemination tree, in which the root of the tree polls the

RSS sources and forwards only new items to the dissemination tree. Using XPORT for disseminating the requested feeds allows RSS sources to receive HTTP requests only from the bootstrap instead from each individual client. Thus, the bandwidth requirements of hosting an RSS feed are decreased. Moreover, since XPORT alleviates the load that would have been presented by many clients, it is reasonable for the bootstrap to poll the RSS source more frequently that any one of the clients would have as traditional RSS clients. In combination with XPORT's push-style distribution, the end result is that clients receive more timely updates while presenting less load on the RSS source. FeedTree [16] possesses a similar structure, though it is unable to perform XPORT's wide variety of optimizations.

5.1 Experimental results

We studied XPORT's performance through real-world data and results from both prototype-based LAN emulation and deployment on PlanetLab. In our PlanetLab experiments we used dissemination trees up 40 randomly chosen PlanetLab sites. For our prototype-based LAN emulation, we also used up to 100 nodes, but controlled network latency and the bandwidth capacities between nodes. We used values obtained from actual PlanetLab measurements, but "replayed" these conditions for multiple experiments in order to obtain repeatable results.

We also created 100 clients and attached them randomly to the XPORT brokers. Each client picks its profile from a set of 700 RSS feeds using the Zipf distribution, with the skew parameter set to 0.97. The total size RSS Feeds was around 19MB, and the average RSS Feed size was 27.7KB. We observed that each transformation of an optimization unit requires around x? secs, and we set our optimization period to y? minutes. The choice of y? minutes is a compromise between rapid adaptivity to network changes and minimizing optimization traffic. The minimal practical interval depends on the metrics being optimized. For example, it takes longer to assess available bandwidth than latency.

We have used XPORT to implement distribution trees that optimize a variety of metrics including total path latency, variance of path latency, bandwidth consumption, bandwidth bottleneck, and total redundant data received at each node. Our experiments demonstrate XPORT's flexibility and effectiveness, as it manages to improve each of these metrics significantly through its local transformations.

Convergence. Figure 5 shows the sum of the path latencies of 20 PlanetLab sites. This is an example of the Type I aggregation. We compare XPORT's performance with the optimal tree, which is the star topology when no constraints are imposed and assuming the triangle inequality and lack of congestion. In the star topology all the nodes are connected directly to the bootstrap node. XPORT starts with a random tree and continuously applies local transformations. The figure shows that, while XPORT starts with lower performance than the star topology, after a small number of transformations our tree converges to the optimal tree.

We also implemented a metric that measures the superfluous data received by the nodes. This is the amount of data each node receives that matches the profiles of its descendants but not the interests of its directly connected clients. A system might want to minimize superfluous data to eliminate disincentives to joining a cooperative system. This is an example of a metric where no aggregation is defined for the node cost. We used this metric to demonstrate XPORT's convergence to the optimal solution and also to show the effectiveness of our transformations.

For this metric we used our RSS feed application on 40 PlanetLab sites. Figure 6 shows three different cases for this metric,

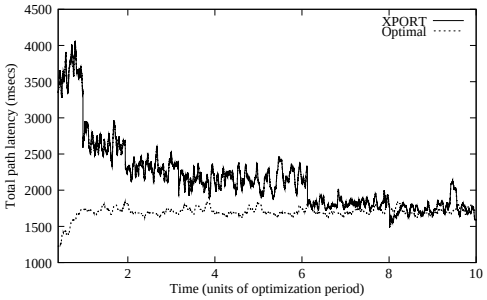

Figure 5: Total network latency of PlanetLab nodes. XPORT converges to the optimal after approximately 8 transformations.

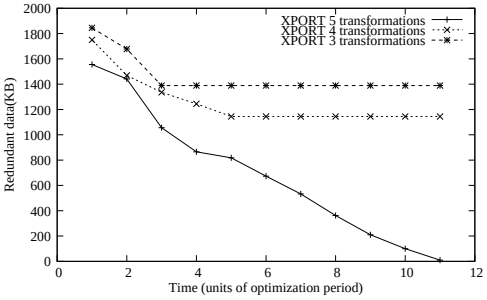

Figure 6: Superfluous incoming data. XPORT converges to the optimal after approximately 11 transformations.

one when all five implemented transformations are applied and two where the transformations are limited to only four and three transformations. We first remove the promote subtree transformation and then the parent-child swap transformation from our optimization framework. The results reveal that in the first case XPORT converged to the optimal solution (where the total redundant data is zero for every node), while for the other two cases the system cost improves but could not converge to the optimal case. Promote subtree is the most beneficial transformation because it allows our tree to converge to the star topology. XPORT managed to improve its performance, even in the absence of this transformation. Finally the results show that the larger the set of transformations, the better performance XPORT achieves.

Adaptivity. While creating a star topology is optimal for the previous metrics, it is not an optimal solution in terms of the resources required from the bootstrap node. To demonstrate this, we run the experiment on the total path latency metric, and included the CPU time for processing and matching the incoming messages on every node on this path. The more profiles that a broker must match, the greater CPU latency that messages will observe when traveling through that broker. In the case of the star topology, the root of the tree has the overhead of matching the incoming messages to the profiles of every node in the system. In the case of XPORT's chosen tree, this processing overhead is distributed across multiple nodes. The bootstrap node performs a few simpler matches on aggregate profiles, one for each child.

Again we started with a random XPORT tree on 20 PlanetLab nodes, which is outperformed at the beginning by the star topology. This difference is due solely to network delays. Earlier experiments indicated that invoking the optimizer at this point would allow XPORT to reconfigure a random tree to match the performance of the star topology. In this experiment we invoke the optimizer only after the bootstrap node begins to fetch and disseminate messages. When the bootstrap node starts fetching the RSS feeds requested by the clients the total path, latency is increased for both

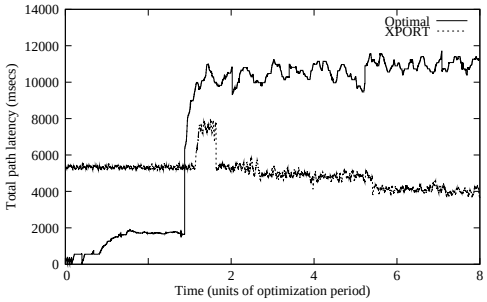

Figure 7: Network latency including cpu overhead. XPORT adapts to increased workload and outperforms the star topology.

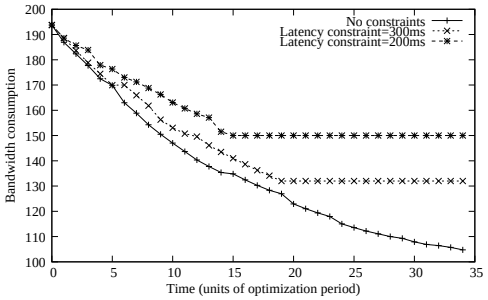

Figure 8: Average bandwidth consumption for different latency constraints.

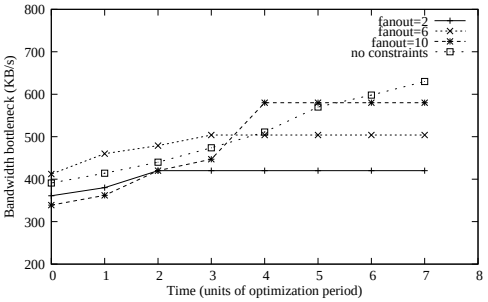

Figure 9: Bandwidth bottleneck under fanout constraints. Applications may impose topology constraints, but these constraints limit XPORT's ability to optimize.

trees as shown in Figure 7. However, the increase is much higher in the case of the star topology, as all matching is performed by the root node. As a result, the random tree begins to outperform the star tree. Moreover, when the optimizer is started, XPORT adapts to the loaded nodes in the system, and after a number of local transformations, obtains a tree that outperforms the original, unloaded random tree. XPORT creates trees that avoid highly loaded nodes, continually moving subtrees to less loaded parents.

Constrained Topologies. We also studied the performance of XPORT when topology constraints are imposed by the application.

We implemented a Type I metric in which the aggregation for the node cost is applied over its children set. We optimized the average outgoing bandwidth over all nodes, where the bandwidth consumption of a node is defined as the sum of the incoming data rate of its children. We run our experiments on 100 nodes on our PlanetLab emulation of the RSS feed dissemination application. Although a better metric might be network utilization, in which the costs of a transmission is multiplied by the number of links in the transmission, it would be difficult to establish an optimal benchmark to compare our tree with. Nonetheless, XPORT could easily optimize such a metric.

As we have defined bandwidth utilization, the optimal topology for this case is again a topology where all nodes with a client profile are directly connected to the root, because each message is emitted by a broker exactly once. We also run experiments with path latency constraints on every node. In the path latency we included the CPU time for processing and matching the incoming messages. This constraint is implemented as aggregation over path.

Figure 8 shows XPORT's performance when no constraints are imposed and two cases where the path latency threshold is set to exceed 300ms and 200ms, respectively. In the first case, XPORT converges to the optimal configuration, after 34 transformations. However, when latency constraints exist, bootstrap cannot accept direct connections from all nodes with a client attached, as that would increase its own CPU latency and the path latency of its descendants more than that threshold. Thus, XPORT cannot converge to the optimal network configuration. It applies a small number of transformations than the no constrained case, and converges to a configuration with higher cost. We note that, more relaxed constraints allow XPORT to perform closer to the optimal case, as more nodes are allowed to connect to the root of the tree.

We also considered a maximum fanout constraint that limited the children of every broker to a constant number. We used XPORT to maximize the lowest bottleneck bandwidth in such a distribution tree. This is a Type IV aggregation. Every node calculates its bottleneck bandwidth, which is the minimum capacity of a link between any two of a broker's ancestors. The system cost is the minimum bottleneck bandwidth over all brokers. The goal of XPORT is to maximize this cost. These results are from our prototype-based LAN emulation on 40 nodes. Figure 9 shows that XPORT identifies bottlenecked portions of the tree and after every transformation increases the maximum bottleneck bandwidth in the system.

We start with very restrictive constraints (fanout of two) and relax them to six, ten and infinite fanout. The results reveal that the stricter the constraints, the fewer transformations XPORT is allowed to perform. For example, when the fanout is set to two, XPORT cannot improve after two optimization periods, while when the fanout is set to six and ten the system improves for three and four periods respectively. These results demonstrate that the more relaxed the constraints imposed by the application the better XPORT can perform.

Table 5 shows similar results for the case of network latency and redundant incoming data metrics. In both cases, when no constraints are imposed XPORT converges to the optimal solution while as the constraints become stricter fewer transformations can be applied and smaller improvement is achieved.

Network traffic. Figure 10 shows the average maintenance traffic and the optimization traffic of every XPORT node for the different metrics we have implemented and various network sizes. *Maintenance traffic* is the data each node exchanges to maintain the dissemination tree, and calculate its own cost. This state is exchanged

Metric	f	t	Performance
Total path latency	2	5	3229ms
	6	8	3036ms
	10	12	2670ms
	∞	14	1691ms
Redundant incoming data	2	4	1088KB
	6	7	461KB
	10	8	304KB
	∞	11	0KB

Table 5: Performance on constrained network (t refers to the number of transformations and f to the fanout constraint). As constraints are relaxed XPORT converges to the optimal tree.

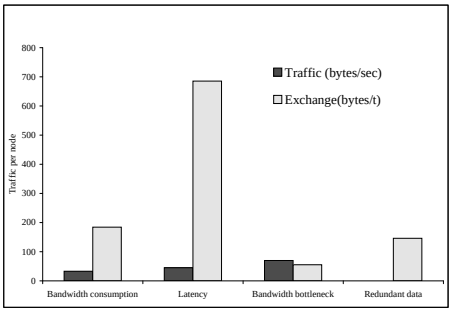

Figure 10: Maintenance and optimization traffic.

every second. *Optimization traffic* is described in Table 4. This state is exchanged between nodes in the optimization unit during an optimization period. The optimization state is also a measure of the optimization overhead.

The results reveal that the maintenance traffic is low and similar for all metrics. Also the maintenance traffic remains the same as the network size increase. We omit the experiments for brevity. Note that for the metric of redundant incoming data there is not cost-related traffic. This is because this metric requires only one level of aggregation as every node's cost is defined as a local value. Thus, node do need to request any information from the parent or child to calculate their cost.

The optimization traffic, while higher than the maintenance traffic, has an acceptable size in all the cases. It is interesting to examine here the difference in the optimization state required by different metrics. In the case of path latency, every node in the network is a critical node. Thus, every node checks for possible transformations of its optimization unit. This implies that all nodes in the system will exchange data with their peers. Moreover, nodes need to send their path latency to nodes in their local neighborhood, like their siblings or nephews in the tree, to allow for various transformations, *e.g.* sibling swap, to be computed.

On the other hand, while optimizing bottleneck bandwidth only the nodes that are affected by the link with the minimum bandwidth capacity try to transform their optimization unit. Thus, only one optimization unit exchanges data, and so the average traffic per node is much smaller. The cases of bandwidth consumption and redundant incoming data are different. In both of these cases, the cost of each node depends on its profile and the merged profile of its children. Thus, the dependence set of a node is smaller, and so fewer nodes exchange state during optimization. This results in less exchanged state for these metrics.

Statistics Approximation We implemented also a metric of Type V, the variance of nodes' path latency. We also implemented a version of the variance metric, where statistics on the node cost and local value are approximated. Our goal in these experiments is to study the trade off between the traffic improvement and the XPORT's performance. Obviously the more frequently nodes broadcast their cost, the better estimation of a transformation's benefit a node can compute, leading to a faster performance improvement. However, as nodes reduce the frequency of their cost broadcast, the maintenance traffic will be reduced.

Our results in 6 demonstrate this behavior. We run our experiments using 100 nodes on our PlanetLab emulation, and computed the variance when nodes participate in every broadcast phase (p=0). Our tree converges to a final configuration after 15 transformations. We used the same tree topology and run the approximated statistics version, where the participation of every node in the broadcast phase is defined by Formula 4, where $w_i = \frac{|V|}{|D_i|}$. In our experiments we varied the period parameter p. The results reveal that even the period is small ($p = 1$), the maintenance traffic can be decreased by 97%, while XPORT converges after 20 transformations to a configuration with cost around 7% more than the non approximated case.

Statistics Period	Performance difference(%)	t	Traffic reduction (%)
$p = 0$	0	15	0
$p = 1$	0.07	20	0.97142
$p = 2$	0.14	26	0.9782
$p = 3$	0.22	31	0.9797

Table 6: Statistics approximation affect on performance, convergence time and network traffic, for the metric of path latency variance. Maintenance traffic is improved significantly.

6. RELATED WORK

Supporting extensibility in systems engineering has often been a key research goal for the benefits brought via modularity and software reuse. In the database community, concepts such as extensibility and declarative specifications have long been the norm as a result of pioneering works such as System R [1] and Starburst [17]. Indeed, the generalization process need not be restricted to the domain of large DBMS, perhaps best exemplified by GiST [9]. GiST provides a framework generalizing the problem of implementing search indexes in a database. In many ways, our work draws its inspiration from GiST, striving to apply the same design principles to distributed data dissemination applications.

Recent efforts from the networking community, such as Click [10], MACEDON [14], and P2 [12] provide examples of systems promoting the advantages of extensibility. Click provides a modular architecture for processing packets in routers using a flow-based configuration specification. MACEDON and P2 both address the challenge of constructing overlay networks by abstracting over commonalities present in the large number of overlay algorithms designed over the last few years.

To the best of our knowledge, we have yet to see extensible data dissemination architectures capable of generalizing over the core dissemination functionality and the system's optimization objective. Existing approaches such as SplitStream [5] and Bullet [11] construct application-level multicast networks that minimize the forwarding load of internal nodes by constructing mesh overlays, thereby enabling clients to receive different data segments from multiple parents in the mesh. ONYX [8] and XRoute [6] introduce content-based publish-subscribe solutions for XML data and XPath-based profiles respectively, and they both focus on using structures for efficiently storing profiles matching them to the incoming data. Siena [3] investigates a publish-subscribe framework for relational data and considers the system's performance from a bandwidth-oriented perspective. By abstracting over the matching functionality, XPORT is able to support both the XPath and relational profiles used by systems such as ONYX and Siena, in addition to supporting a superset of the optimization metrics considered by these systems.

Also closely related are those approaches that use the concept of local transformations to perform continuous adaptive optimization of the dissemination tree [2, 18]. These systems attempt to optimize a specific metric, as opposed to the general optimization framework provided by XPORT. Finally, AMMO [15] provides a similar framework for constructing an adaptive multi-metric overlay networks. Their metric-independent framework focuses on minimizing the sum of a performance metric defined over all the overlay edges of the dissemination tree. Compared to AMMO, XPORT's model is more extensible, since we allow a wider variety of cost

functions and a generic means to combine them.

7. CONCLUSIONS AND FUTURE WORK

XPORT explores routing tree extensibility in the context of profile-based data dissemination systems. Our work is largely motivated by the growing number of medium-large scale dissemination-based applications and services; addressing the needs of this broad application domain requires a robust and flexible dissemination infrastructure that is also highly extensible and customizable. XPORT is a step towards building such an infrastructure.

We have built an initial XPORT prototype system and will soon deploy it on PlanetLab, where it will be running as a profile-based feed dissemination service. This experience will allow us to better debug our system and gather user profiles for further experimentation.

As future work, we have a full agenda. First, we will explore how to extend our tree-based overlays to more general mesh-based topologies, which will further improve the efficiency and reliability of dissemination but complicate optimization. Second, we will extend XPORT to also serve as a data collection system. The interesting challenge here is the seamless integration and optimization of both collection and dissemination tasks. Finally, we will investigate extensible profile specification languages to ease the specification of complex, stateful profiles.

8. REFERENCES

[1] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. Gray, P. P. Griffiths, W. F. K. III, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: Relational approach to database management. *ACM Transactions on Database Systems*, 1(2):97–137, 1976.

[2] S. Banerjee, C. Kommareddy, K. Kar, S. Bhattacharjee, and S. Khuller. Construction of an efficient overlay multicast infrastructure for real-time applications. In *INFOCOM*, 2003.

[3] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, Aug. 2001.

[4] A. Carzaniga and A. L. Wolf. Forwarding in a content-based network. In *SIGCOMM*, 2003.

[5] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. I. T. Rowstron, and A. Singh. Splitstream: High-bandwidth multicast in cooperative environments. In *SOSP*, 2003.

[6] R. Chand and P. Felber. Scalable Protocol for Content-Based Routing in Overlay Networks. In *NCA*, 2003.

[7] Y. Diao and M. J. Franklin. Query processing for high-volume xml message brokering. In *VLDB*, 2003.

[8] Y. Diao, S. Rizvi, and M. J. Franklin. Towards an Internet-Scale XML Dissemination Service. In *VLDB*, 2004.

[9] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *VLDB*, 1995.

[10] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[11] D. Kostic, A. Rodriguez, J. R. Albrecht, and A. Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *SOSP*, 2003.

[12] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *SOSP*, 2005.

[13] O. Papaemmanouil and U. Çetintemel. SemCast: Semantic Multicast for Content-based Data Dissemination. In *ICDE*, 2005.

[14] A. Rodriguez, C. Killian, S. Bhat, D. Kostic, and A. Vahdat. Macedon: Methodology for automatically creating, evaluating, and designing overlay networks. In *NSDI*, 2004.

[15] A. Rodriguez, D. Kostic, and A. Vahdat. Scalability in adaptive multi-metric overlays. In *ICDCS*, 2004.

[16] D. Sandler, A. Mislove, A. Post, and P. Druschel. Feedtree: Sharing web micronews with peer-to-peer event notification. In *IPTPS*, Ithaca, New York, Feb. 2005.

[17] P. M. Schwarz, W. Chang, J. C. Freytag, G. M. Lohman, J. McPherson, C. Mohan, and H. Pirahesh. Extensibility in the starburst database system. In *OODBSs*, 1986.

[18] Y. Zhou, B. C. Ooi, K.-L. Tan, and F. Yu. Adaptive reorganization of coherency-preserving dissemination tree for streaming data. In *ICDE*, 2006.