Extensible Optimization in Overlay Dissemination Trees*

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

We introduce XPORT, a profile-driven distributed data dissemination system that supports an extensible set of data types, profile types, 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 filtering, profile aggregation, and optimization logic. The clean separation between the "plumbing" and "application" enables the system to uniformly support disparate dissemination-based applications.

We first provide an overview of the basic XPORT model and architecture. We then describe in detail an extensible optimization framework, based on a two-level aggregation model, that facilitates easy specification of a wide range of commonly used performance goals. We discuss distributed tree transformation protocols that allow XPORT to iteratively optimize its operation to achieve these goals under changing network and application conditions. Finally, we demonstrate the flexibility and the effectiveness of XPORT using real-world data and experimental results obtained from both prototype-based LAN emulation and deployment on PlanetLab.

1. INTRODUCTION

XPORT (eXtensible Profile-driven Overlay Routing Trees) is a generic profile-driven distributed data dissemination system. It is designed to provide the core dissemination infrastructure for a growing set of dissemination-based applications and services, including web feed dissemination (RSS/Atom), multicast-based content distribution, massively multiplayer network games, stock ticker distribution, and large-scale distributed collaborative applications.

Dissemination-based applications often exhibit diverse application logic and performance requirements. At the same time, they all require several *common* core facilities, which include dissemination overlay construction, maintenance and optimization, (content-based) routing logic, and membership management. These applications are often developed from scratch, requiring substantial effort and investment to "get it right" for each specific case. In contrast to the existing approaches that provide point solutions to point problems, XPORT's goal is to develop an application-agnostic solution that can be easily customized and extended for a specific target application through a small number of methods that encapsulate application-specific behavior and optimizations.

Extensibility is the central design consideration for our system, which supports an extensible set of data and profiles types, and optimization metrics. Specifically, XPORT supports two types of extensibility. *Profile-related extensibility* refers to the ability to easily accommodate new data and profile types, and is key to supporting diverse applications. *Cost-related extensibility* refers to the ability to express application-specific performance goals, and allows applications to define their own criterion of an efficient and effective dissemination system. Given application-defined data types, profile types and performance metrics, XPORT automatically builds, maintains and optimizes an overlay dissemination tree consisting of the available broker machines in the system.

The main focus of this paper is on the design, implementation, and evaluation of XPORT's basic optimization framework that uses a novel two-level aggregation model to define system cost. The first level computes the cost of each node as an aggregation of metrics gathered from the node's local neighborhood. The second level computes the system cost by aggregating the node costs. The model allows for the uniform specification of many commonly used performance measures, as well as new ones, through combinations of different aggregation functions and local metrics.

During run time, the system iteratively applies tree transformations in order to converge to a minimal cost configuration, optionally subject to constraints (*e.g.*, "minimize the total bandwidth consumption in the system while ensuring that the dissemination latencies do not exceed 100ms"). The transformations are guided by a set of transformation rules, including primitive rules defined by XPORT and application-defined composite rules obtained through the composition of the primitive ones.

XPORT uses a tree-oriented cost model to estimate the benefit of each potential tree reconfiguration. With the knowledge of the semantics of the aggregation functions and transformation rules, the system derives and automatically collects the nodes states and statistics required. XPORT also employs an approximation technique that adjusts the statistics sampling rate at each node on the basis of the node's estimated contribution to the system cost.

Our contributions in this paper can be summarized as follows:

1. We introduce the basic design and architecture of XPORT, which, to the best of our knowledge, is the first dissemination system that provides profile and cost *extensibility*.
2. We present a tree-oriented optimization framework that uses a novel metric-independent multi-level aggregation model to express system cost metrics. The framework includes a grammar that facilitates the specification of a large set of performance measures and constraints, including ones that involve multiple metrics, as well as an extensible set of transformation rules.
3. We describe distributed iterative optimization protocols that efficiently implement the optimization framework using

*This work is supported in part by the NSF under grants IIS-0325838 and IIS-0448284.

†Author performed the work while he was a summer intern at Brown University.

cost-function specific optimization techniques.

4. 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's optimization approach.

We begin by introducing the system's API in Section 2. We introduce the optimization framework and provide a detailed discussion of its non-operational aspects in Section 3. We describe the basic XPORT architecture and its run-time behavior 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. XPORT API

In the first part of this section we motivate our API, by discussing the common characteristics of dissemination systems. We then describe the methods an application needs to define to express its native data types, profile types and performance goals.

2.1 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 accordingly motivate the general methods used by XPORT.

Profile-driven dissemination systems 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 producers generate data by *publishing* and consumers *subscribe* to data through their profiles. The underlying dissemination system is responsible for delivering to clients data matching their profiles.

The dissemination infrastructure consists of a set of nodes (often called *brokers*) organized into an overlay network. Here on, we will use the terms brokers and nodes interchangeably. This network usually consists of one or more dissemination trees [3, 8, 15]. Clients subscribe by forwarding their profiles to a broker. These 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 created, a broker can now forward incoming data to the subset of its children that is interested in receiving the data, instead of forwarding each data message to all its children, thereby eliminating the "flooding" problem. This routing scheme works by *matching* each data message with the routing table entries that represent the aggregated profile for each subtree.

Depending on the application's data types and the complexity of its profiles, dissemination systems may use their own algorithms and *indexing structures* for efficiently storing profiles on every broker and matching incoming data against them. ONYX [8] uses YFilter [7] for matching XPath profiles, whereas SIENA [3] 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).

2.2 Application-defined methods

Based on the main functionality of data dissemination systems, 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

These methods describe how the matching of data and profiles will be performed. Optionally, the application can specify how profiles should be stored, indexed and maintained at each node.

- **match**(m**,** p): Given a data message m and a profile p, it returns true if m matches p, or false otherwise.
- **merge**(p**,** q): Given two profiles p and q, it returns a more general profile covering p and q. This function can merge profiles received from clients or children, reducing the routing state maintained in a node and 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 profile p to the index
  - **remove**(p): removes a profile p from the index
  - **match**(m**,** ind): Given a data message m and a profile index ind, it returns the set of profiles matching m.

By default, XPORT stores every new profile as a separate routing entry, and uses a disjunction operator for profile merging.

2.2.2 Cost-related methods

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

aggregate (function, value, set).

Similarly, applications can also specify constraints for each node. Figure 1 shows the grammar for defining the performance criteria.

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 client profiles, its children's aggregated profiles. These are denoted in the grammar by the terms `METRICS` and `STATE`, respectively. We now describe our grammar and the two-level aggregation model in a top-down manner, starting from its second level.

System cost. An application defines the system performance metric, which we refer to as the *system cost*. This is an aggregation of the node cost values over all nodes (or clients):

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

In order to generalize the aggregation technique and make the presentation more succinct, we categorize the aggregation functions into three classes: (i) *additive functions* (`SUM`, `AVERAGE`), (ii) *bottleneck functions* (`MIN`, `MAX`) and (iii) *holistic functions* (`VARIANCE`, `STANDARD DEVIATION`, `PRODUCT`). This categorization is based on the state required by the nodes for optimization purposes. In particular, for the holistic functions, nodes can identify beneficial optimizations by estimating changes on the cost

```
<SYSTEM COST SET>: brokers | clients | brokers–clients
<SYSTEM COST FUNCTION>:MIN | MAX | SUM | AVERAGE |
                       PRODUCT | VARIANCE | STD
<SYSTEM COST>: aggregate(<SYSTEM COST FUNCTION>, <NODE COST>,
                       <SYSTEM COST SET>)
<NODE COST SET>: path | children
<NODE COST FUNCTION>: MIN | MAX | SUM | AVERAGE
<NODE COST>: aggregate(<NODE COST FUNCTION>, <LOCAL VALUE>,
                       <NODE COST SET>) |
             <LOCAL VALUE>|
             g({<NODE COST>})
<LOCAL VALUE>:f(<STATE>|<METRICS>,  link | node, <NODE COST SET>) |
              f( link | node, <NODE COST SET>)
<CONSTR METRIC>: <SYSTEM COST> |<NODE COST>
<CONSTRAINT>: <CONSTR METRIC> <OP> threshold
<OP>: < | > | <= | >= | !=
<METRICS>: path latency | incoming data rate | ...
<STATE>: profile set | merged profile | ...
```

Figure 1: **Cost metric grammar.**

of the nodes affected by the optimization, while for the additive and bottleneck functions, nodes need to estimate changes on some aggregated state for these nodes. Moreover, the state required for the additive functions can be restricted even further (see Section 3.2). Our definition permits applications to define a variety of system cost measures, like minimum bandwidth capacity, total bandwidth consumption, average path latency, etc.

Node cost. The node cost can be defined as (i) an application-defined local metric, (iii) a combination of metrics defined as the node cost, or (ii) an aggregation of the local metrics of some neighboring nodes. In the last case the aggregation function is:

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

The *node cost function* can be either an additive function or a bottleneck function. *Node cost set* defines which neighbors' local metrics we will aggregate. It could be either the nodes on the path to the root (referred as *aggregation over path*), or the 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 per node can be defined as an aggregation over children; it is the sum of the incoming data to each child. XPORT can also create a multi-metric overlay network by allowing applications to specify the node cost as a combination of multiple optimization metrics, *e.g.*, the product of latency and bandwidth.

Local value. The application also defines the node's local metric, which we call the *local value*. Applications can either use a built-in metric or provide a method for computing this metric. This method can also have as input some of the predefined state variables or metrics, *e.g.*, the expected incoming data rate, can be 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 its links with its neighbors (*e.g.*, latency to the parent). This option is specified by the parameter *node* or *link* in the metric implementation method. The exact link on which the local value will be calculated is determined by the `NODE COST SET` term of the grammar. If this term is set to *path*, *i.e.*, we have an aggregation over path, then the local value is measured on the link to the node's parent. If the node cost is an aggregation over children, then the term is set to *children*, and the local value is measured on the links to the children. Finally, our definition allows the local value to be defined as a combination of multiple metrics.

Constraints. Constraints are specified as:

Aggregation Type	System Cost Function	Node Cost Function	Example Metrics
Type I	Additive	Additive	average path latency
Type II		Bottleneck	total path bandwidth bottleneck
Type III	Bottleneck	Additive	maximum path latency
Type IV		Bottleneck	min path bandwidth bottleneck
Type V	Holistic	Additive	variance of path latency
Type VI		Bottleneck	variance of bandwidth bottleneck

Table 1: **Two-level aggregation examples.**

constraint (metric, operator, threshold)

Constraints are basically defined in the same way as the system cost, *i.e.*, following the two-level aggregation model, with an additional threshold for the constrained metric. For example, an application might want to impose an upper bound on the path latency of every node, which is one-level aggregation over path. Similarly, a dissemination system might try to guarantee a lower bound of the maximum path latency. This is a two-level aggregation of the path latency over all the nodes system. XPORT customizes its functionality and optimization framework to respect these constraints.

Table 1 shows the different aggregation function combinations along with some example metrics.

2.3 Cost metric examples

In this section, we provide some example metrics. We start with the *average path latency*. Here, every node measures the link latency to its parent and adds this to the path latency of its parent:

system cost = aggregate (`AVERAGE`, path latency, `BROKERS`)
path latency = aggregate (`SUM`, link latency, `PATH`)

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

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 redundant incoming data*. In this case, the application aims to minimize the undesired data each broker receives and forwards. This is the data the broker is not interested in receiving itself, but has to do so in order to forward it to its descendants who are interested.

system cost = aggregate (`SUM`, superfluous data, `BROKERS`)
superfluous data = (extraIncoming(), `PARENT`)

The function *extraIncoming()* estimates the difference between the incoming data rate and the matching rate of the client profiles.

Our last example uses a combination of metrics for the node cost. We define the node cost as the bandwidth-delay product of its path. This metric provides an estimation of the amount of data currently in transit on the path. The performance goal is to minimize the average product over all nodes.

system cost = aggregate (`AVERAGE`, node cost, `BROKERS`)
node cost = (path latency $\times$ bandwidth)
path latency = aggregate (`SUM`, link latency, `PATH`)
bandwidth = aggregate (`MIN`, link bandwidth, `PATH`)

3. OPTIMIZATION FRAMEWORK

XPORT strives to create overlay trees that minimize application-specified cost functions. Periodically, XPORT modifies the tree structure using local transformations to adapt to time-varying network or workload conditions. We informally define a *local transformation* as one that requires interactions among only the "nearby" brokers on the overlay tree. In our current implementation, these

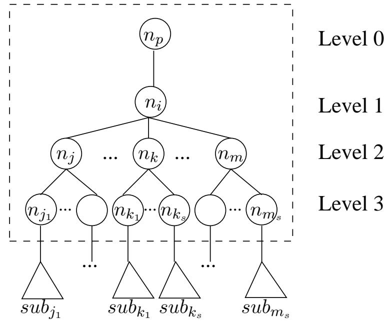

Figure 2: Optimization unit of n_i. Nodes inside the dashed box belong to the unit.

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. We refer to these nodes collectively as the *optimization unit* of n_i.

A local transformation is *transparent* outside its optimization unit; *i.e.*, the transformation does not affect the optimization unit's interface with the rest of the network. This implies that the path from n_p to the root and the subtrees below the last level of the unit will not be affected in terms of their topology and (merged) profiles. Thus, both the parent of n_p and the brokers at the last level will continue forwarding the same data to the same nodes, as they did before the transformation. The only nodes that might experience a change in their connections and profiles are those within the optimization unit. However, any node's cost might be affected by a transformation. Keeping the topological and profile-related effects of our transformations local reduces the cost of network reconfiguration, as fewer nodes are affected by each transformation.

3.1 Local transformations

A brute-force approach for identifying the best transformation of an optimization unit would be to consider all possible reconfigurations of the unit's structure. There are two main drawbacks of this approach. The first is the exponential number of configurations that need to be considered. The second is the increased communication overhead—exchanging information to quantify the benefit of each transformation may be prohibitively expensive. Instead of performing an exhaustive search of all configurations, XPORT limits its search to a smaller set of "promising" transformations. This set contains a number of built-in *primitive transformations* as well as other composite transformations defined by the application. Our experimental results show that a small number of well-chosen transformations can be very effective while incurring low overhead.

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

Child demotion. 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.

Child promotion. This transformation moves a node n_{j_1} 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.

The transformation set of XPORT is extensible. Applications can define their own *composite* transformations by using the primitive ones. Allowing composite transformations is important as they improve convergence times and also could prevent XPORT from settling in local minimums. In our implementation, we defined the following composite transformations.

Transformation	Definition
Child demotion	$demote(n_k, n_j)$
Child promotion	$promote(n_{j_1})$
Subtree promotion (n_j)	$promote(n_{j_s})^*$
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 transformation rules and how they are composed to create complex rules ('→' indicates the ordering between operations; '*' indicates that the operation will be performed repeatedly for all nodes whose parent is specified as the first parameter. The naming of nodes refers to their position in the original unit).

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

Parent-child swap. In this transformation, the owner of the optimization unit n_i and its child n_j swap positions, without moving the subtree of n_j (Figure 3(d)). This will force every subtree previously under n_i or n_j to have a different parent. Parent-child swap is derived by combining child promotion and demotion.

Subtree migration. In this case the subtree of a node n_j migrates under its sibling node n_k (Figure 3(e)). Node n_j remains with no children, while the subtree of n_j increases by the subtree of n_k. Subtree migration can be derived from the two primitive transformations, by first promoting all the children of n_j under n_i and then demoting the same nodes under n_k.

Sibling swap. Here, two siblings n_k and n_j swap positions. This will change the root node of their subtrees, 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. Similar tree transformations were also used by previous work [2, 20].

We note that optimization units can be defined differently; in particular, they can be extended to more than three levels. Extending the optimization scope in this manner increases the flexibility of the system, as it facilitates a larger number of more powerful transformations. On the downside, such an extension also increases the maintenance traffic and the size of the state nodes need to collect and maintain. Investigating the cost vs. effectiveness tradeoffs for varying optimization scope sizes is an interesting issue that is outside the scope of this paper.

3.2 System Cost Improvement

The goal of the local transformations is to improve the overall system cost. XPORT calculates the cost benefit of every candidate transformation and applies the best one. Since the system cost is an aggregation of the node costs, the exhaustive approach for quantifying this cost benefit is to estimate the cost of every node after the transformation and aggregate them to get the new system cost. This approach may have prohibitively high communication overhead. However, XPORT can avoid this overhead as it understands the semantics of the aggregation functions. This knowledge allows XPORT to efficiently quantify the cost effect of a transformation. Furthermore, XPORT can automatically identify the state required

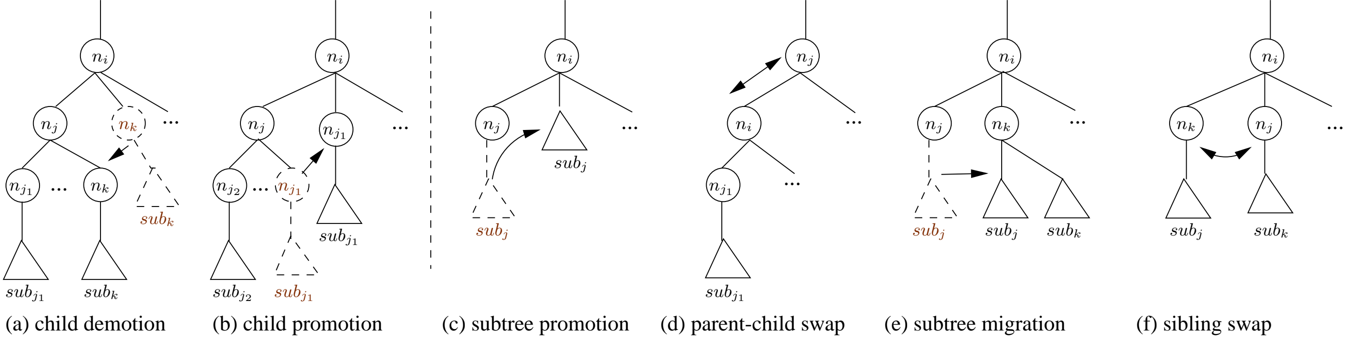

Figure 3: Local transformation rules: (a) and (b) are primitive and (c)-(f) are composite transformations.

by each node, as well as the information to be exchanged among nodes during the optimization. In the rest of the section, we describe this approach in detail.

3.2.1 Quantifying a transformation's benefit

In this section we provide the general equations that estimate the cost benefit of a transformation. We start with some definitions and continue with our metric-independent equations for the cost 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 *(respectively the nodes of the path from* n_i *to the root) when* $cost_i$ *is calculated as aggregation over path (respectively aggregation over children). We refer to members of* D_i *as the* dependents *of* n_i.

For example, for aggregation over path, an increase on the latency of the link between n_i and its parent will increase the path latency of all nodes 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 different data messages downstream.

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 include all nodes in the network. Since the node cost is defined as a local value, XPORT limits, whenever possible, the dependence set based on the definition of this metric. For example, if the local value is defined as a function of the profile of n_i, then the dependents of n_i are the nodes of its path, since a change in this profile could affect only these nodes.

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 function. We denote a change of* $cost_i$ *that affects* $cost(D_i)$ *as* $\Delta cost(D_i)$, *and the new dependence set cost as* $cost'(D_i)$.

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

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

Let U_i denote the set of nodes in the optimization unit of n_i and S_i denote the set of nodes in the last level of this unit, when the node cost is an aggregation over path (*e.g.*, nodes of Level 3 in Figure 2). For an aggregation over children, S_i is the root of the optimization unit. We refer to the union of the dependents of all nodes in S_i as the *dependence set* L_i *of the unit of* n_i. That is, the set of nodes that do not belong in the optimization unit of n_i but may be affected by its transformation.

Finally, we denote $\Delta cost_i$ as the cost change of n_i due to a transformation in its unit, $cost'_i$ as its new cost value, and b_i as the benefit of the transformation with respect to the system cost. We provide now the equations that quantify the cost benefit of a transformation in the optimization unit of n_i.

Additive functions. Consider the case where the system cost is the SUM of the node costs. Then, b_i is the sum of the *cost change* of only the nodes affected by the transformation, *i.e.*, nodes inside the unit and the unit's dependents:

$$b_i = \sum_{j \in U_i} (cost'_j - cost_j) + \sum_{k \in S_i} \Delta cost(D_k) \qquad (1)$$

If the AVERAGE function is used, then b_i is divided by the number of nodes in the system. This general equation holds for every transformation. However, depending on the transformation, many of its terms are zero, so simpler equations can be obtained. Examples of these equations are given in Table 3.

Bottleneck functions. We assume that the system cost function is MIN. Then n_i estimates the new minimum cost among all nodes. This can be computed by aggregating only the new cost of the affected nodes and the minimum cost of all nodes not affected by the transformation. If c denotes the current system cost, then the benefit of the transformation is:

$$b_i = c - \min_{j \in U_i, k \in S_i} \{cost'_j, cost'(D_k), \min_{m \notin U_i, m \notin S_i} \{cost_m\}\} \qquad (2)$$

where $cost'(D_k) = cost(D_k) + \Delta cost(D_k)$. To estimate the minimum cost of all nodes not affected by the transformation state of constant size is required at every node. We will provide details in the Section 3.2.2.

Holistic functions. In this case n_i calculates the new cost of every node affected, and estimates the difference with the current cost, using its estimations of all the nodes costs. Thus, for the VARIANCE function:

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

where $\overline{cost'}$ is the average node cost after the transformation and V is the set of nodes.

Note that some holistic functions can be evaluated (or approximated) more efficiently than the naive approach presented here by leveraging the semantics of the specific function under consideration.

Child demotion	$(cost'_k - cost_k) + \Delta cost(D_k)$
Child promotion	$(cost'_{j_1} - cost_{j_1}) + \Delta cost(D_{j_1})$
Subtree promotion	$\sum_{k \in children_j} \Delta cost(D_k) + \sum_{k \in children_j} (cost'_k - cost_k)$
Subtree migration	$\sum_{k \in children_j} \Delta cost(D_k) + \sum_{k \in children_j} (cost'_k - cost_k)$
Sibling swap	$\sum_{s \in children_k \cup children_j} \Delta cost(D_s) + \sum_{s \in children_k \cup children_j} (cost'_s - cost_s)$

Table 3: Simplified cost equations for local transformations (when the system cost function is SUM).

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. Obviously, increasing the size of the optimization unit will simply increase the number of terms in Equations 1, 2 and 3, adapting them to include the extra nodes.

Defining new transformations is simple from the user perspective. The user simply defines which transformations will be combined along with the desired parameters. From the system perspective, defining a new transformation requires the definition of the equation that quantifies its cost benefit. This equation can be derived by aggregating, with the system cost function, the equations for each of the transformations that define the composite one. For example, it is straightforward to see that, for the subtree promotion, the equation is simply the sum of the equations for the promote child operation, over all children of node n_j. For more complicated composite transformations, like sibling swap, more compact equations will be derived, as many of their terms cancel out. Providing compact equations implies less traffic during optimization, due to the smaller state exchanged among nodes in the optimization unit.

3.2.2 *Quantifying cost changes on nodes*

The equations that quantify the benefit of a transformation require estimating the cost effect on the unit's nodes and on its dependents. Moreover, for the additive and bottleneck functions the effect on the dependents can be estimated by calculating the affect on their aggregated cost, *i.e.*, the dependence set cost. XPORT strives to incur the minimum communication overhead when estimating these cost changes. Therefore, it limits the communication among nodes of the same unit and thus avoids any metadata exchange with the unit's dependents, as they lie outside the optimization unit. Instead, nodes maintain some metadata for their unit's dependents. We refer to this state as the *transformation state*.

XPORT exploits the semantics of the aggregation functions to identify both the minimum transformation state required and to derive generic equations that quantify the cost change of the unit's dependents. We provide the details of our approach in the following paragraphs, for different types of node cost functions. We note that if no aggregation is used for the node cost, XPORT uses Equations 1, 2 and 3 to quantify the result of a transformation by estimating the new cost of the dependents of the unit.

Additive node cost functions. For the purpose of illustration, we focus on the case of the SUM function and an aggregation over path to the root. An example metric is path latency. Our results are similar for the case of the AVERAGE function and aggregation over children. In this case, a change of a node's cost (*i.e.*, path latency to the root) will incur the same change on the costs of all its dependents (*i.e.*, every node in its subtree). Given this, we describe the different cases of system cost functions, when the cost of n_i changes by $\Delta cost_i$. Note that for the additive and bottleneck functions we need to compute the change of the dependence set cost of a node n_i, while for the holistic functions we need to estimate the cost change of each dependent of n_i. Since each dependent's cost change is $\Delta cost_i$, the total change of the dependence set cost for the SUM system cost function is :

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

Thus, node n_i needs to maintain only the size of its dependence set.

Consider the case of bottleneck functions. Here, the system cost function is either MIN or MAX. Thus, we are only interested in the change on the minimum (or maximum) node cost among the dependents. Thus:

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

Moreover, based on Equation 2, every node simply needs to maintain an estimation of the current system cost and its dependence set cost.

For the holistic functions, the cost change of every dependent is defined as:

$$\Delta cost_j = \Delta cost_i$$

For this case, every node stores an estimation of the current system cost, as well as the cost of every node in the system. Given these equations, node n_i can estimate the benefit of a transformation using Equations 1, 2 and 3.

Bottleneck node cost functions. We focus here on the MIN aggregation function, for the purpose of illustrating our ideas. Our results can be easily extended for the MAX function. Again, we assume that an aggregation over path is used for the definition of the node cost, 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 minimum bandwidth capacity over all links on its path. Here on, we will refer to this bandwidth capacity as the bottleneck value of the node, and the link with this capacity as the bottleneck link. For simplicity, we will present our approach with respect to this metric.

Changing 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 of the cost of n_i may affect the cost of every dependent n_j (*i.e.*, all nodes in its subtree). This effect depends on the links lying between the nodes n_i and n_j, since there may be similar bottlenecks between them. The following definition allows us to identify these links.

DEFINITION 3. *The* cost of a dependent n_j of n_i 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, using the node cost function. Moreover, the aggregation of n_i's local value and $h_i(j), \forall n_j \in D_i$, is referred to as the* minimum local value of n_i, g_i.

For the bandwidth bottleneck metric, $h_i(j)$ is the minimum bandwidth capacity link between n_i and its descendant n_j, while g_i is the minimum bandwidth capacity of the links in the subtree of n_i, including the local value of n_i.

Each node maintains the above metric for all its dependents. In the case of additive functions, we can reduce the amount of state by storing only the unique $h_i(j)$ values along with the frequency for each distinct value:

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

We mentioned in the previous section, that in order for a node to estimate the impact of a transformation on the system cost it needs

to know the minimum cost of all system nodes min_{net}, except the ones affected by a trasformation. This implies that every node needs to maintain the minimun cost of all nodes that do not belong in its dependents set. This value can be easily calculated by using the set T_i, defined above.

We now describe the equations based on which n_i can calculate a change in its dependence set cost, when its cost change is $\Delta cost_i$ and we start with the case of an additive function for the system cost.

We assume again the system cost function is SUM and we are interested in the total cost change over all the dependents. We distinguish two cases; one where the cost of n_i decreases and one where 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

$$\gamma = \sum_{\alpha} (\min\{cost_i, \alpha\} - cost'_i) \times \lambda, \text{ s.t. } (\lambda, \alpha) \in T_i, \alpha \geq cost'_i$$

In the first case, n_i and its dependents share the same bottleneck link, so their cost is affected by $\Delta cost_i$. 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 calculates their total cost change.

In the second case, $cost_i$ increases:

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

where

$$\omega = \sum_{\alpha} \Delta cost_i \times \lambda + \sum_{\alpha'} (\alpha' - cost_i) \times \lambda'$$
$$\text{s.t. } (\lambda, \alpha) \in T_i, \alpha \geq cost'_i \text{ and } (\lambda', \alpha') \in T_i, \alpha' < cost'_i$$

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 value after the transformation. Given this estimation of $\Delta cost(D_i)$, n_i uses Equation 1 to estimate the benefit of a transformation. Thus, the state required for this case is simply the size of the dependence set and the set T_i.

For the bottleneck functions, 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:

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

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

Similarly if $cost_i$ increases then:

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

Similarly for the holistic functions, we distinguish two cases; one when the $cost_i$ decreases and one when it increases. For the first case, the cost change is:

$$\Delta cost_j = \begin{cases} \Delta cost_i & g_i \geq cost_i \\ \gamma' & \text{otherwise} \end{cases}$$

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

Table 4: Transformation state T and optimization state O for node n_i (V is the set of nodes in the system).

where

$$\gamma' = \min_{n_j \in D_i, h_i(j) \leq cost'_i} \{cost_i, h_i(j)\} - cost'_i$$

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:

$$\Delta cost_j = \begin{cases} \Delta cost_i & g_i > cost_i \text{ and } g_i > cost'_i \\ \omega' & \text{otherwise} \end{cases}$$

where

$$\omega' = cost'_i - h_i(j)$$

3.2.3 Multi-metric cost functions

XPORT can create multi-metric overlay trees by defining either the node cost or the local value of a node as a combination of multiple metrics. For both of these cases, the individual metrics used in the definition of the combined metric are calculated independently. Each node combines these metrics, based on the application-specified function, to obtain the final local value or node cost, respectively. Moreover, for multi-metric node costs, the optimization procedure is somewhat different. For each transformation, the node calculates the effect of the transformation on its dependents separately for each individual metric. It then combines these metrics to derive the final impact on its dependents' costs. Since there are no restrictions on the combination functions, the total benefit of a transformation is calculated similarly to the holistic functions, *i.e.*, each node uses its estimation of the new node costs to calculate the new system cost and compares it with the current cost value.

3.2.4 Statistics approximation

In the previous section, we argued that maintaining state for the dependents of the optimization unit of n_i (*e.g.*, dependence set cost) allows n_i to quantify the benefit of a transformation. In order for n_i to calculate this state, its peers need to periodically broadcast some metadata. This metadata includes the cost and local value of each node. For the additive and bottleneck system cost functions, n_i needs to collect this data only from its dependents, while in the case of the holistic functions, it needs to know the cost of every node in the system. To reduce the high overhead of these broadcasts, we vary their frequency.

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

$$w_i \times p \quad (4)$$

where $p > 1$ is a predefined constant.

The weight of a node depends on the node cost function. We distinguish two cases; the first one refers to the additive functions. In this case, the larger the dependence set of a node is, the more nodes it can affect if its own local value changes. To reflect a node's impact on the system cost, we set the weight of n_i to be $w_i = \frac{|V|}{|D_i|}$. An example metric is a node's path latency. In this case, changes on the links closer to the root affect more nodes than the links closer to the leaf nodes.

The second case is for the bottleneck functions. Here, the closer the local value of n_i is to its dependents' cost, the more likely it is to become their new bottleneck value (assuming no drastic changes on the local values of the nodes). Thus, we set the weight of n_i to $w_i = local_value_i - g_i$, where g_i represents the minimum (or maximum) bottleneck value of all the dependents of n_i. An example metric is the bandwidth bottleneck of a node, where a change on a link's capacity could potentially affect the bandwidth of all its descendants. ants.

4. RUN-TIME FUNCTIONALITY

In this section, we describe the run time functionality of XPORT. We start by describing the basic system model and architecture. We then describe the details of the distributed optimization protocol.

4.1 System model

XPORT consists of a set of nodes organized into an application-level overlay tree. In order to join the system, a new node selects an existing node as its parent. No specific attempt to find a "good" parent is made at this point with the expectation that subsequent optimization steps will move the node to a more optimal and valid (wrt. to the constraint, if any) network location.

The high-level usage is based on the publish/subscribe paradigm, where sources publish their data and clients express their data interests through their profiles. A client connects to an XPORT node and registers its profile. Nodes add a new profile p as a new routing entry, optionally indexing it as described in Section 2. The new profile is merged with the existing ones, using the merge function if it exists. The profile is then propagated up the tree towards the root, until it reaches either a node with a more general "covering" profile or eventually the root.

Upon the receipt of a data message m, a broker checks all the entries in its routing table to determine whether m should be forwarded to a downstream broker (or a client) using the appropriate matching function.

Node architecture. The high-level architecture of an XPORT node is shown in Figure 4. Applications customize two main system components. The first component is the *data/profile handler* that is responsible for storing, indexing and maintaining profiles as well as matching them against incoming data messages. The *optimizer* identifies and applies network transformations on the basis of application-specified performance criteria and constraints. The tree transformations to be performed are given to the *connection manager*, which establishes and manages the node's connections with its parent and children. Both the optimizer and the profile/data handler communicate with the node's *router*, which handles all the data

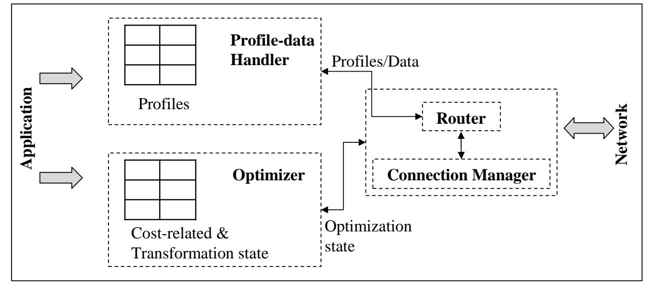

Figure 4: XPORT's high-level node architecture.

and metadata communication.

Node State. Each node maintains four state types: *profile-based*, *cost-based*, *transformation*, and *optimization* state. The profile-based state includes the profiles the node receives from its children and clients, and the merged profile it derives from them. Thus, the size of the profile-related state for node n_i is $O(|children_i|)$. This state resides in the profile/data handler. The cost-based state includes the local metric value of the node and its cost. This state is of constant size, $O(1)$. The transformation state refers to the data 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 and resides in the optimizer. Optimization and transformation states and their size estimations are given in Table 4.

4.2 Distributed optimization protocols

4.2.1 Bottleneck optimization

XPORT uses a *bottleneck-based* approach for optimization in which the system focuses only on *effective* transformations that have the potential to reduce the overall system cost. Other transformations, even though they might yield smaller local costs, are ignored. For example, if the optimization goal is to minimize the maximum CPU load in the system, then XPORT will focus solely on the most loaded node and attempt to decrease its cost.

To implement this approach, we rely on the notion of a *critical node*. A node is considered critical if a change in its cost may potentially affect the system performance. Similarly, we define a *critical optimization unit* as a unit that may affect the cost of a critical node, and, thus, by definition the system cost.

In XPORT, optimization proceeds over *optimization periods*. At each optimization period, each node that owns a critical optimization unit exchanges data (*i.e.*, optimization state) with the nodes in its unit, and quantifies the benefit of the candidate transformations. It then identifies the most effective transformation and sends it to the root of the tree. The root simply identifies the transformation with the maximum expected benefit and informs the selected unit about the new configuration to which it should switch. This bottleneck-based approach ensures cost improvement in every optimization period, assuming that at least one beneficial transformation is identified. In addition to the candidate transformations, the system also considers (with low probability) an additional random transformation to avoid getting stuck in a local optimum.

During tree reorganization, nodes maintain their old connections while they establish their new connections. This approach allows the nodes to keep receiving data from the current tree during optimization periods. To ensure the correctness of tree reorganization and avoid losing messages, however, care must be taken regarding when the nodes should switch to the new sub-tree. One solution is to wait until all connections are set up and then use a distributed

protocol to make the switch in a coordinated fashion, starting from the root of the new sub-tree proceeding downstream.

XPORT implements an alternative approach that is based on the use of *sequence numbers* and a TCP-like window-based message request and retransmit scheme implemented at the application level. In this approach, each message injected to the system is assigned a unique monotonically-increasing sequence number by the root. Nodes simply cache the messages they receive until they run out of space. Whenever a node establishes a new connection (either during reorganization or after a disconnection/failure), it requests from its parent the messages that it has not yet received, which it can determine on the basis of the sequence numbers it has seen. If the parent node does not have those messages in its cache, it will forward the request to its own parent and the process will iterate. This approach not only eliminates the need for coordinated switching but also allows XPORT to deal uniformly with other problematic cases such as failures and temporary disconnections. The details of the protocol are outside the scope of this paper.

An alternative to bottleneck-based optimization is an approach we refer to as *opportunistic* optimization. If the optimization goal is to minimize the maximum CPU load, then the opportunistic approach will attempt to reduce the load of the most loaded node *in each optimization unit*, as opposed to the bottleneck-based approach that will only consider the critical unit(s). Even though the opportunistic approach will lead to many "useless" transformations (wrt. the system cost metric), it also has the potential to indirectly lead the system to a globally good configuration over time. Investigating the tradeoffs between the two approaches is an interesting future research direction.

4.2.2 Concurrent transformations

A related issue is the choice of the number of optimization units that can be optimized concurrently. Strictly serializing transformations is easier to reason about and implement, as we do not need to consider potentially negatively interfering concurrent transformations across multiple optimization units. At the same time, serialization often slows down the converge rate to a minimal cost configuration.

We now discuss how XPORT facilitates multiple concurrent transformations by reasoning about the scope and semantics of the transformations. This reasoning is based on the notion of *independence* for optimization units and transformations.

DEFINITION 4. *The optimization units of n_i and n_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 the optimization units of n_i and n_j, respectively. Furthermore, let $trans_set_i$ 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 the optimization units of n_i and n_j are independent, or*

$$(trans_set_i \cap trans_set_j) = \oslash.$$

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

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

If two transformations are applied on independent units, then they cannot negatively interfere with each other. Otherwise, one of the transformations might potentially affect the cost of some of the nodes in the other optimization unit. Thus, we need a way to quantify the benefit of parallel independent transformations when their corresponding optimization units overlap. We denote this benefit as b_{ij} and estimate it as follows.

Additive functions. Assuming the system cost function is `SUM`, then, if the two units are independent, the total benefit is the sum of the benefit of each transformation. For overlapping units, the total benefit is calculated in the same way since each node's change affects its dependents equally. Thus:

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

Bottleneck functions. If the two units are independent and the system cost function used is `MIN`, then the total benefit is:

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

When the units overlap, assuming that n_i is in a higher level in the tree than n_j, the total benefit of the parallel transformations is:

$$b_{ij} = c - \min_{m \in U_j, k \in S_j} \{cost'_j + x, cost'_m + x, cost'(D_k), min_{net}\}$$

where $cost'(D_k) = cost(D_k) + \Delta cost(D_k)$ and x is the cost increase of n_k's first ancestor that belongs in the unit of n_i but not the unit of n_j. Also we define min_{net} as :

$$min_{net} = \min_{y \notin U_i, y \notin S_i, y \notin U_j, y \notin S_j} cost_y$$

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

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

5. PERFORMANCE EVALUATION

We implemented an initial XPORT prototype [14] in Java. For experimentation purposes, we also built an RSS feed dissemination application using the XPORT API and deployed it across the PlanetLab testbed.

For this application, XPORT automatically builds an overlay tree, where the root of the tree polls the RSS sources and forwards only new items to the other nodes in the tree. Using XPORT for disseminating the requested feeds allows RSS sources to receive HTTP requests only from the root instead of each individual client. Thus, the bandwidth requirements of hosting an RSS feed decreases. Moreover, since XPORT alleviates the load that would have been presented by many clients, it is reasonable for the root 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 [18] possesses a similar structure, though it is unable to perform XPORT's wide variety of optimizations.

5.1 Experimental results

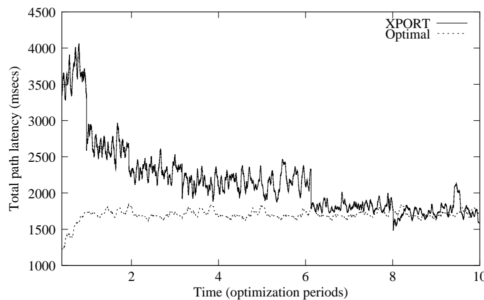

Figure 5: Total network latency of PlanetLab nodes. XPORT converges to the optimal after approximately 8 transformations.

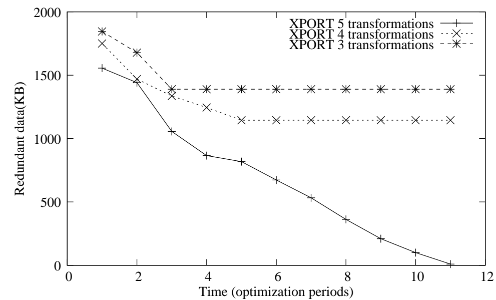

Figure 6: Superfluous incoming data per broker on 40 PlanetLab nodes. XPORT converges to the optimal after approximately 11 transformations.

We studied XPORT's performance through real-world RSS data and results from both prototype-based LAN emulation and deployment in the PlanetLab testbed. In our PlanetLab experiments, we used dissemination trees with up to 40 randomly chosen PlanetLab sites. For our prototype-based LAN emulation, we used up to 100 nodes, but artificially controlled the network latency and bandwidth capacities between nodes. We obtained these metrics 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 nodes. 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 of RSS feeds was around 19MB, and the average RSS feed size was 27.7KB. For the experiments, we set our optimization period to two minutes. This choice 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, average bandwidth consumption, bandwidth bottleneck, and total received redundant data. 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 network path latencies of 20 PlanetLab sites. This is an example of an aggregation using additive functions for both the node and system cost. 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 root of the tree. 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 total redundant data received by the nodes. This is a sum of the amount of data each node receives that matches the profiles of its descendants but not the interests of its directly connected clients. An application 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, one when all five composite transformations are applied and two where the optimizer used only four and three composite 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. Moreover, the larger the set of transformations, the better performance XPORT achieves.

Adaptivity. While creating a star topology is optimal for the previous cases, it is not an optimal solution in terms of the resources required from the root 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 a node must match and the higher its fanout, 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 case of XPORT's chosen tree, this processing overhead is distributed across multiple nodes.

Again we started with a random tree on 20 PlanetLab sites, which is outperformed at the beginning by the star topology. This difference is due solely to network delays. 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 root node has fetched the RSS feeds and completed their dissemination. When the root starts fetching the RSS feeds requested, the total path latency increases for both 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 starts, 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 constraints are imposed by the application. To demonstrate this case, we use a metric where a node's cost is aggregated over its children's local values. 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 of its children. For this metric, we run experiments with path latency constraints on every node. In the path latency we included the CPU

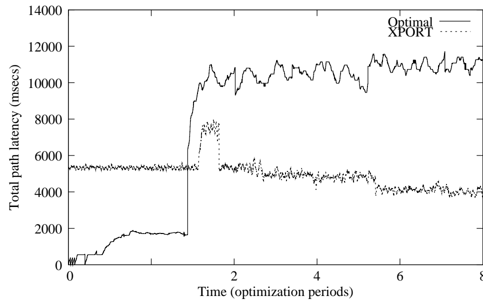

Figure 7: Network latency including CPU overhead. XPORT adapts to increased workload and outperforms the optimal topology.

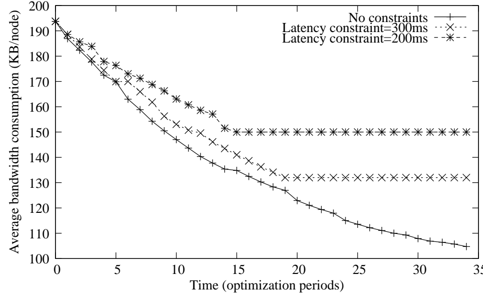

Figure 8: Average bandwidth consumption for various latency constraints.

time for processing and matching the incoming messages. This constraint is implemented as aggregation over path.

We run our experiments on 100 nodes in our LAN emulation environment. 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. 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. Figure 8 shows XPORT's performance when no constraints are imposed and two cases where the path latency threshold is set to 300ms and 200ms. In the first case, XPORT converges to the optimal configuration. However, when latency constraints exist, the root 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, violating the constraint. Although XPORT cannot converge to the optimal network configuration, more relaxed constraints allow the system 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 an aggregation that uses the bottleneck function `MIN` for the node and system cost. 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 critical portions of the tree and after every transformation increases the minimum bottleneck bandwidth in the system.

We start with a maximum fanout of two, a very restrictive constraint, and relax the constraint to six, ten and infinite fanout. The results reveal that the stricter the constraints, the fewer transforma-

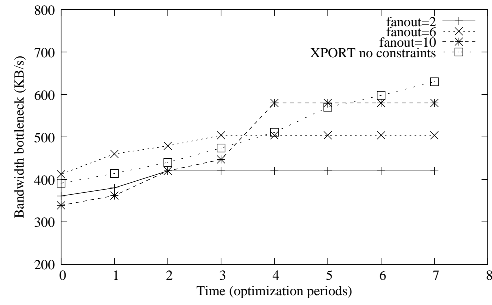

Figure 9: Bandwidth bottleneck under fanout constraints. Applications impose topology constraints, which may limit XPORT's ability to optimize.

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 a 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.

tions XPORT can perform. For example, when the fanout is set to two, XPORT cannot improve beyond two optimization periods. When the fanout is set to six and ten, the system improves for three and four periods, respectively. Table 5 shows similar results for network latency and redundant data metrics. In both cases, when no constraints are imposed, XPORT can converge to the optimal solution. As the constraints become tighter, fewer transformations can be applied and smaller improvement is achieved.

Network traffic. Figure 10 shows the average maintenance traffic and the optimization traffic for the different metrics for a network of 40 nodes. *Maintenance traffic* is the data each node exchanges with its parent and children, to calculate its own cost. This state is exchanged in specific time periods (*maintenance period*). *Optimization traffic* is the data needed to estimate the cost of candidate transformations and is metric specific. This state is exchanged between nodes within the same optimization unit during an optimization period. Intuitively, the optimization state is a measure of the optimization overhead.

The results reveal that the maintenance traffic is low and similar for all metrics. Note that, for the redundant data metric there is no maintenance traffic. This is because this metric requires only one level of aggregation, as each node's cost is defined as a local value. Thus, nodes do not need to request any information from their parents or children to calculate their costs.

The optimization traffic, while higher in most cases than the maintenance traffic (per period), has an acceptable size. It is interesting to examine the difference in the traffic required by different metrics. In the case of the total path latency metric, each node in the network is essentially a critical node. Thus, all nodes check for possible transformations of their optimization units. This implies that all nodes in the system will exchange data within their optimization units.

On the other hand, while optimizing bottleneck bandwidth, only the nodes that are affected by the link with the minimum bandwidth capacity attempt to transform their optimization units. Thus, only those optimization units that contain the bottleneck edge will

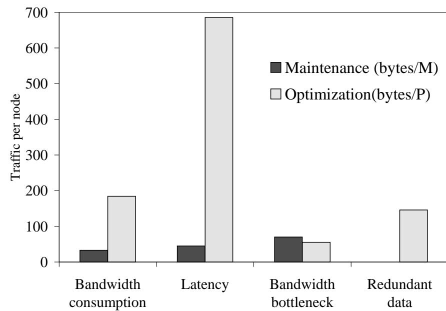

Figure 10: Maintenance and optimization traffic (S = Maintenance period, P= optimization period).

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

Table 6: Statistics approximation effects on performance, convergence time t and network traffic, when minimizing path latency variance. The percentage values indicate relative differences over the non-approximated case.

exchange data. As a result, the average traffic per node is much smaller. The cases of bandwidth consumption and redundant incoming data are yet 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 for optimization.

Statistics Approximation. We now study the tradeoff between traffic improvement and system performance when approximating statistics. For this purpose, we ran an experiment where the optimization goal is to minimize the variance of the path latencies across all the nodes in the system to ensure service fairness. We also ran another version of the algorithm where statistics on the node cost and local values are approximated. We expect that the more frequently nodes broadcast their cost values, the better cost estimations they can compute, leading to more effective transformations and thus faster convergence. Of course, if nodes broadcast their costs more frequently, the maintenance traffic will be higher.

The results in Table 6 clearly demonstrate this tradeoff. We ran our experiments using 100 nodes in the LAN emulation environment, and computed the variance when all nodes participate in every broadcast phase. In this case, the tree converged to its final configuration after 15 transformations. The average bandwidth consumption was approximately 2KB per node per period. We used the same tree topology and ran 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 the experiments, we also varied the period parameter p. The results reveal that even for small period values ($p = 1$), approximation reduces the maintenance traffic by 97%. At the same time, it takes 20 transformations for the approximated approach to converge to a configuration with a cost value that is approximately only 7% more than that of the non-approximated case. The results for larger period values reveal similar benefits.

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 [19]. Indeed, the generalization process need not be restricted to the domain of large DBMSs, 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 [16], 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 optimization objectives. 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, in addition to supporting a superset of the optimization metrics considered by these systems.

Closely related are also those approaches that use the concept of local transformations to perform continuous adaptive optimization of the dissemination tree [2, 20]. These systems attempt to optimize a specific metric, as opposed to the general optimization framework provided by XPORT. Finally, AMMO [17] 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. It is largely motivated by a growing set of medium-large scale dissemination-based applications and services. Addressing the requirements of this broad application domain requires robust and flexible software infrastructures that are also highly extensible and customizable. Our work is a step towards building such an infrastructure.

We implemented an initial XPORT prototype [14] on which we have built two applications: a peer-to-peer RSS feed dissemination service and a networked multiplayer game. We are currently in the process of deploying these applications on PlanetLab. This experience will allow us to better debug our system and gather real user profiles for further experimentation.

As future work, we have a full agenda. First, we will extend XPORT to also serve as a data collection system. The interesting challenge here is the seamless integration and combined optimization of the collection and dissemination tasks. Second, we will investigate extensible profile specification languages to ease the specification of complex, stateful profiles (e.g., aggregates, joins).

Finally, 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.

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, Y. Ahmad, U. Cetintemel, J. Jannotti, and Y. Yildirim. Extensible profile-driven data dissemination. Technical report, Brown University, CS-06-05, 2006.

[14] O. Papaemmanouil, Y. Ahmad, U. Cetintemel, J. Jannotti, and Y. Yildirim. XPORT: Extensible profile-driven overlay routing trees (demonstration). In *SIGMOD*, 2006.

[15] O. Papaemmanouil and U. Çetintemel. Semcast: Semantic multicast for content-based data dissemination. In *ICDE*, 2005.

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

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

[18] 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.

[19] 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 *OODBS*, 1986.

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