Application-aware Overlay Networks for Data Dissemination [*]

Olga Papaemmanouil Yanif Ahmad Uğur Çetintemel John Jannotti
Brown University, Department of Computer Science, Providence, RI USA
E-mail: {olga,yna,ugur,jj}@cs.brown.edu

Abstract

XPORT (eXtensible Profile-driven Overlay Routing Trees) is a generic data dissemination system that supports an extensible set of data types and profiles, and an optimization framework that facilitates easy specification of a wide range of useful performance goals. XPORT implements a tree-based overlay network, which can be customized per application using a small number of methods that encapsulate application-specific data-profile matching, profile aggregation, and cost 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 dissemination and multicast-based content distribution.

In this short paper, we provide a high-level overview of XPORT. We also discuss its current implementation status, applications we built using XPORT, and some preliminary experimental results from the prototype. We finalize the paper by summarizing our future directions.

1. Introduction

XPORT is a generic profile-driven distributed data collection and dissemination system. Extensibility is the central design consideration for XPORT. Our model allows an extensible set of data types, profiles and optimization goals. XPORT is designed to support the core data dissemination infrastructure for a growing set of dissemination-based applications and services, including web feed dissemination (RSS/Atom), massively multiplayer network games, multicast-based content distribution, large-scale collaborative applications, 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 underlying 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 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-profile, cost methods and optimization rules, XPORT automatically builds and maintains an overlay dissemination network consisting of the available broker machines in the system. The system iteratively applies local optimizing transformations to adapt to changes in the network and workload conditions, and to incrementally converge to an overlay configuration that is "optimal" (as defined by the application).

In this paper, we provide an overview of XPORT. We begin by introducing the system's API, which allows the specification of application-defined methods that encapsulate the behavior of the system (Section 2). We continue with XPORT's optimization framework, which uses a novel two-level aggregation model to define system cost (Section 3). 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. We also describe our prototype and the applications we built, as well as present some preliminary implementation results (Section 4). Finally, we discuss our plans for future extensions to XPORT (Section 5).

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

2. XPORT API

We start by motivating XPORT's basic API by discussing the common characteristics of dissemination-based applications. We then continue by describing the methods an application needs to define to express its "native" data and profile types and performance targets.

2.1. Background: Dissemination-based Applications

To introduce XPORT's API, it is helpful to review profile-driven data dissemination applications in simple terms. The goal here is to highlight the common functionality in these applications to 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 [6, 11, 16]. 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 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 of the broker.

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 [11] uses YFilter [10] for matching, whereas SIENA [6] uses a custom index [7] for storing and matching relational profiles.

Different dissemination-based systems and applications can have widely varying efficiency and effectiveness 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 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 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.

Profile-related methods. These methods specify how profiles are processed, stored, indexed and maintained at each broker of the system. XPORT's API provides a **merge(**p**,** q**)** function that allows an application to define how two profiles p and q are merged to a more general profile covering these profiles. It also allows applications to integrate an index structure by specifying three methods: (i) an **init()** method for declaring and initializing the index structure; (ii) an **add(**p**)** method that adds a new profile p to the index; and (iii) a **remove(**p**)** method that removes a profile p from the index.

Finally, message matching is specified by the method **match(**m**,** p**)** that returns true if a message m matches a profile p or false otherwise. If an indexing structure is used then a method **match(**m**,** ind**)**, can be defined. This method returns the set of profiles in the index ind that match the message m.

Cost-related methods. XPORT allows applications to specify their own performance criteria, based on which the dissemination network is created. XPORT 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 cost metric. We define the **node cost** as:

$$aggregate(aggregation\ function, value,\\ aggregation\ set)$$

The *aggregation function* can be *min*, *max*, *sum* or *product*. The *value* is a metric calculated on every node, over the node's local neighborhood (*e.g.*, link latency to its parent or children), or the node itself (*e.g.*, CPU latency). It may either be a predefined performance metric provided by XPORT or defined by the application by providing a method to calculate it. For example, XPORT provides a built-in method for calculating the link latency between nodes. However, a user can also provide its own method to measure this metric. The above formula allows applications to

System cost function	**Node cost function**	**Optimization metric example**
sum	sum	total outgoing bandwidth consumption
average	sum	average path latency
average	product	average path reliability
average	min	average path bandwidth
max	sum	maximum path latency
min	product	minimum path reliability
min	min	min path bandwidth

Table 1. Two-level aggregation examples.

define a large set of metrics, used frequently for the evaluation of dissemination-based systems, such as path latency, outgoing bandwidth consumption, or even matching overhead.

The second level of aggregation computes the **system cost** by aggregating the node costs:

$$aggregate(aggregation\ function, node\ cost, aggregation\ set)$$

This *aggregation function* can be *min*, *max*, *sum* or *average*. XPORT allows aggregation to be computed over a number of entities, on which the node cost is defined, *e.g.*, nodes, paths, clients, etc. This allows applications to define a large variety of system cost measures, like maximum path latency, total bandwidth consumption, or average matching overhead.

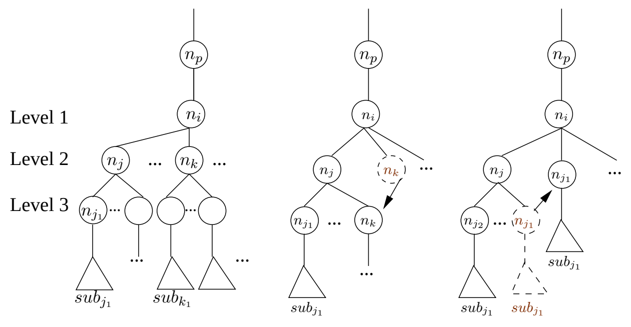
Similarly, applications can also define **constraints** as:

$$constraint(metric, operator, threshold)$$

This method allows a threshold to be specified for some local metric of every node, *e.g.*, maximum fanout of a node, maximum profiles stored, etc. XPORT customizes its functionality and optimization framework to respect these constraints. Table 1 shows some example metrics specified using the two-level aggregation model.

3. Extensible Optimization

XPORT strives to create dissemination trees that respect application-specific performance metrics. Periodically, it modifies the overlay tree using local transformations to adapt to changing network and workload conditions affecting the system's performance. We define a local transformation as one that requires interactions between "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, children and grandchildren, as shown in Figure 1(a). We call these nodes collectively the *optimization unit* of broker n_i. Our local transformations do not affect the optimization unit's interface with the rest of the network. Keeping the 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.



(a) original tree (b) demote child (c) promote child

Figure 1. XPORT's primitive transformations.

XPORT provides two *primitive transformations*. These transformations are *child demotion* and *child promotion* (shown in Figure 1(b) and 1(c), respectively). We now explain these transformations with respect to the 3-level optimization unit shown in Figure 1(a).

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 while 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 under its grandparent n_i. This increases the number of subtrees of n_i while leaving n_j with one less subtree.

The goal of the local transformations is to improve the overall system cost. For every transformation, XPORT calculates the cost change on every node affected by the transformation. Since XPORT understands the semantics of the aggregation functions, it can automatically quantify the effects of a transformation. This is achieved by the use of general formulas that hold regardless of the specific aggregation functions being used. 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. Both the state requirements and the communication costs are $O(1)$ for each node, for most of the aggregation functions supported by our model.

Extending the transformation set. XPORT's transformation set is also extensible. Applications can define their own *composite* transformations by using the primitive ones. An example composite transformation is the promotion of a subtree, which is a sequence of child promotions for all the children of a node. Composite transformations are important as they help avoid being stuck in a local optimum.

Extending the scope of optimization units. As mentioned earlier, our optimization unit includes nodes within three levels of each other. However, the optimization unit can easily be extended to include nodes at most n-levels from each other. This extension allows an application to define more complex composite transformations. However, while it introduces new opportunities for optimization, it also increases the state requirements at each node and the overall communication cost.

3.1. Optimization approaches

XPORT employs two approaches for optimization; *bottleneck* and *opportunistic* optimization. The bottleneck approach tries to optimize the system cost function for the entire system, while the opportunistic one optimizes the cost function locally for each optimization unit. 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, while the opportunistic approach will reduce the workload of the most loaded node in each optimization unit.

For bottleneck optimization, we rely on the notion of a *critical node*. A node is considered critical if its 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 opportunistic approach, all nodes participate in the optimization period, while in the bottleneck approach only nodes inside a critical optimization unit will attempt to optimize. In both cases, the most effective transformation is identified and applied. Note that, the bottleneck approach guarantees cost improvement in every optimization period, if at least one beneficial transformation is identified. However, although the opportunistic approach does not provide such guarantees, it can also indirectly lead the system to a globally near-optimal configuration.

Another optimization issue is the number of optimization units that can be optimized in parallel. The main benefit of the 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 convergence to the “optimal” topology. In our current implementation, we allow parallel transformations to take place in non-overlapping optimization units. In the future, we will investigate concurrent transformations for overlapping units.

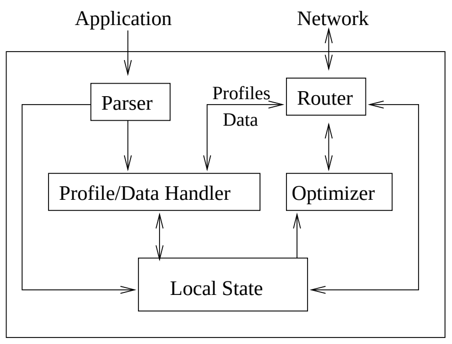


Figure 2. XPORT node architecture.

4. Current Status

We developed an initial XPORT prototype and built two related data dissemination applications on top of it. The first application disseminates RSS feeds and the second allows distribution of results based on queries to the Google Scholar search engine [1]. In this section, we briefly describe the main components of our prototype and the applications we built. We also present some preliminary experimental results.

4.1. Prototype

Our current prototype is implemented in Java. In our implementation, nodes are organized into a single dissemination tree rooted at a *bootstrap* node. The bootstrap node is responsible for periodically polling the sources and forwarding any new messages to the dissemination tree.

The basic architecture of an XPORT broker node is shown in Figure 2. Application-defined methods are given to each node’s *parser*, which customizes various system components to meet the application-specific profiles, data types and performance metrics. The *profile/data 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 meta-data 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 meta-data.

4.2. Applications

RSS is a family of XML file formats for web syndication commonly used by news websites and weblogs. Site owners publish their site content via an XML file called an *RSS feed*. Users can request these feeds through a *feed reader*, a client application for subscribing to the feeds, periodically polling the RSS sources and presenting the returned data.

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 through their feed reader, and the proxy registers their subscription to an XPORT node. The bootstrap node 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. FeedTree [19] possesses a similar structure.

In our second application, we used queries to the Google Scholar search engine as our client profiles. Google Scholar provides a way to broadly search for scholarly literature. In our implementation, a user can search across many disciplines and sources by specifying a set of attribute values like author, document title, keywords and publication date. Results are transformed to an RSS format and pushed to the users, which can then read them through their feed readers. Clients with similar profiles can share the same RSS feed. The advantage of our application is that it provides a pushed-based interface to a traditional pull-based application, allowing users to automatically get new results as they become available.

4.3. Preliminary Results

We deployed our implementation of the RSS feed application across the PlanetLab testbed and ran some initial experiments. We 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. The total size of RSS Feeds was around 19MB. The bootstrap node periodically downloads these RSS feeds and disseminates them down the tree.

The first metric we experimented with is the total path latency. Figure 3 shows the sum of the path latencies of 20 PlanetLab sites. 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. Our preliminary results show that, while XPORT starts with lower performance than the star topology, after a small number of transformations, our tree converges to the optimal tree. In the future, we will investigate various optimization metrics, like bandwidth consumption and CPU latency and compare XPORT performance to the optimal dissemination topology.

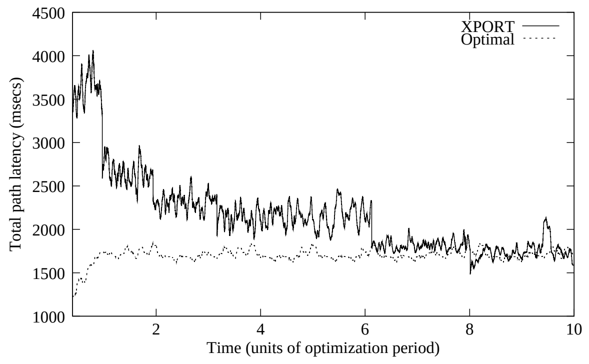


Figure 3. Total network latency of PlanetLab nodes. XPORT converges to the optimal after approximately 8 transformations.

5. XPORT Extensions

5.1. Stateful subscriptions

An important goal of XPORT is to support profile extensibility. Our current implementation allows stateless profiles to be evaluated independently over each incoming message. However, many applications require the ability to handle stateful subscriptions, e.g., "drop similar messages if they arrive with less than 10 seconds". In order to support such applications, we plan to extend XPORT's API.

Stateful operations require messages to be stored for some period of time, e.g., if we need to eliminate duplicates with at most x seconds difference in their arrival time, we need to store the incoming messages for this time frame. Thus, we plan to support a more expressive *match* function, which facilitates state maintenance. There are two approaches we are currently investigating. The first maintains this state globally for the application, applying these operations for all the messages entering the system. This will be beneficial for cases where the operations have the same definition and semantics for all clients, *e.g.*, messages with temperature values with less than 5% difference and 10 seconds apart are considered the same for a specific weather alert application.

In the second approach, every subscriber can define its own stateful operation as part of its profile, e.g., some clients consider temperature readings with 5% difference as the same if they appeared in the last 20 seconds while others do so if the messages appeared in the last 10 seconds. In this case, state needs to be maintained per profile in our system.

5.2. Customizing delivered messages

Apart from stateful subscriptions, many applications desire subscriptions to transform the structure and values of incoming messages. A simple example would be subscriptions mapping message values to a different value domain. XML streaming applications where subscriptions are expressed as XPath queries [10, 11] also transform original messages before they get delievered to the subscribers. In this case, XPath queries are applied on incoming XML documents, and can return specific elements of the input document.

To this end, we plan to further extend our *match* function to allow data transformations. In our current API, this function simply takes as input a single message and identifies if it matches a given profile or not. This function will be extended to provide also the output message to be sent downstream.

5.3. Integrating collection and dissemination

Until now, we have described XPORT as an extensible data dissemination infrastructure. Prior to dissemination, XPORT is first responsible for gathering data from a potentially very large number of data sources. Currently, XPORT's collection functionality is implemented by the bootstrap node periodically polling the sources. We propose the construction of an overlay topology to support efficient data collection, as a dual to the tree for dissemination.

As our first extension, we plan on distributing the burden of data collection from a single root node, to a tree-shaped topology. XPORT will extend its framework to the collection task by defining a set of primitive transformations on the collection tree, just as in the dissemination tree. One example of a transformation is the ability to split the list of sources a node gathers data from and assign this workload to a parent or child node.

By combining data collection and dissemination into a single framework, XPORT will be able to take advantage of transformations specifically for end-to-end optimization of the dataflow path. One example of such an optimization technique is short-circuiting. Short-circuiting involves adding a direct overlay link between an interior node of the collection tree and an interior node of the dissemination tree. This technique may be applied in scenarios where the data collected by the node in the collection tree covers the data requested by the profiles in the dissemination tree.

Furthermore, XPORT may have to handle the use of multiple optimization metrics, one for collection, and another for dissemination. In these scenarios, we will have to apply transformations over combined metrics, or attempt to individually optimize metrics through the use of multiple overlay trees. We return to the topic of optimizing with multiple trees in Section 5.5.

XPORT will also support the notion of source advertisements that specify the content a source produces. Adverts are the collection tree's equivalent of a profile, and XPORT will use these adverts to perform processing tasks on the data as it is gathered. Examples of these tasks include presentation (e.g., translation of the message to another language), aggregation and duplicate elimination, amongst others. We intend to augment XPORT's core API to support collection related methods, as complements of the profile related methods, to allow the application developer to customize these tasks. Other extensions include supporting the equivalent of a matching index for data collection purposes. This collection index could be used to efficiently determine which data sources to poll and when to poll them.

5.4. Supporting high-level profile languages

Currently, XPORT requires the user to ”fill in” all relevant API methods using custom code. While it is rather easy to specify simple profiles types (such as basic predicates) this way, the specification of more sophisticated ”stateful” profiles (e.g., ”give me only those news items that don't look like any I have received during the last two days”) may be overly difficult, especially when dealing with methods such as merge() (see Section 2). Therefore, to simplify the job of the user, we plan to provide a ”native” profile language that consists of a small number of powerful stream-oriented operators with well-known semantics. Such a language will make it easy to express complex user profiles, while allowing XPORT to automatically derive and implement the associated API methods based on the knowledge of the operator semantics.

To this end, we plan to use a subset of the dataflow language of Aurora [3], which includes stream-oriented operators that can perform simple filtering and merging, as well as complex time-window-based aggregation and correlation over data streams. As such, users will be able to express their profiles using either a graphical data-flow notation or a textual notation, both of which are already supported by Aurora and Borealis [2]. In addition to the benefits mentioned above, expressing profiles in this manner will allow XPORT to use Aurora's high-performance data-flow execution engine to perform low-latency message matching and forwarding.

5.5. Supporting multiple collection and dissemination trees

The current model of XPORT allows the construction of a single dissemination tree. However, previous work [8, 14, 16] has shown that an overlay mesh can achieve fundamentally higher efficiency and reliability compared to a single tree. Thus, we intend to extend our system to support the construction and maintenance of optimized meshes. This will allow XPORT to improve significantly on metrics like throughput, bandwidth consumption and create better customized trees for its subscribers.

XPORT is currently limited to tree topologies by two design decisions. First, cost metrics are evaluated along the single path between a node and the root. Second, optimization units are defined by tree semantics (a node, its children, and grandchildren). Together, these design decisions simplify the computation of new costs after a local transformation is performed. In order to maintain the benefit of these decisions, yet support the advantages of mesh topologies, we will build meshes from multiple overlapping trees.

XPORT's cost computation and optimization unit will require extensions to accommodate this change. To evaluate the cost of multiple trees, XPORT will evaluate each tree independently using our two-level aggregation model. Then, a third aggregation level will be introduced to compute the mesh cost by combining the scores for each tree.

XPORT's optimization unit will also be extended to allow for optimizations across trees. We are considering a number of approaches that would allow for transformations that are sufficiently local to permit incremental cost recomputation following a change, yet allow for transformations that affect multiple trees.

The concern is that a transformation that affects a single optimization unit in one tree might involve nodes that are widely scattered in another tree. It would be difficult to efficiently assess the effects of swapping two nodes that are located far from each other in a tree. With this in mind, transformations will only affect *topology* in one tree at a time. However, a topology change in one tree may still affect the cost of another tree. For example, a node may suddenly be asked to perform more forwarding, affecting its available bandwidth and its CPU load. However, these smaller effects are readily quantified in the "other" tree. The total effect of a change can be computed as the sum of the effect caused by the topology change in one tree and the effects caused in each of the other trees by the change in available resources that came as a byproduct of the topology change.

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 [4] and Starburst [20]. Indeed, the generalization process need not be restricted to the domain of large DBMS, perhaps best exemplified by GiST [12]. 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 [13], MACEDON [17], and P2 [15] 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 [8] and Bullet [14] 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 [11] and XRoute [9] 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 [6] 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 [5, 21]. These systems attempt to optimize a specific metric, as opposed to the general optimization framework provided by XPORT. Finally, AMMO [18] provides a similar framework for constructing an adaptive multi-metric overlay networks. Their metric-independent framework focuses on mimimizing 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

XPORT explores overlay routing tree design and extensibility in the context of profile-based data dissemination systems. Our work is largely motivated by the growing number of medium-to-large scale dissemination-based applications and services. Addressing the needs of this broad application domain requires a robust and flexible dissemination infrastructure that is application aware, highly extensible and easily customizable per application. 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 a profile-based feed dissemination service. This experience will allow us to better debug our system and gather user profiles for further experimentation.

References

[1] Google Scholar, http://scholar.google.com/.

[2] D. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The Design of the Borealis Stream Processing Engine. In *CIDR*, 2005.

[3] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *VLDB Journal*, 12(2), 2003.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] R. Chand and P. Felber. Scalable protocol for content-based routing in overlay networks. In *NCA*, 2003.

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

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

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

[13] 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.

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

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

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

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

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

[19] 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.

[20] 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.

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