An Extensible Overlay Infrastructure for Wide-Area Stream Processing and Dissemination

by

Olga Papaemmanouil

B. S., University of Patras, Patras, Greece, 1999

M. S., Athens University of Economics and Business, Athens, Greece, 2001

M. S., Brown University, Providence, RI, USA, 2004

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island

May 2009

This dissertation by Olga Papaemmanouil is accepted in its present form by the Department of Computer Science as satisfying the dissertation requirement for the degree of Doctor of Philosophy.

Date ______________ ________________________________
Uğur Çetintemel, Director

Recommended to the Graduate Council

Date ______________ ________________________________
John Jannotti, Reader

Date ______________ ________________________________
Stan Zdonik, Reader

Approved by the Graduate Council

Date ______________ ________________________________

Dean of the Graduate School

Vita

Olga Papaemmanouil was born on November 22, 1976 in Alexandria, Imathia, Greece. She received her undergraduate degree in Computer Engineering and Informatics at the University of Patras, Greece in 1999. In 2001, she received her Sc.M. in Information Systems at the University of Economics and Business, Athens, Greece. She worked as a software engineer for the Athens Stock Exchange untill 2002. She then joined the graduate program in Brown University, where she received her Sc.M. in Computer Science at Brown University, in 2004. She completed her Ph.D. in Computer Science in Brown University in 2008. She has been a recipient of the Paris Kanellakis Fellowship.

Acknowledgments

The first person I would like to thank is my advisor, Ugur Cetintemel. Ugur's positive attitude inspired me from the first time I met him, in my visit at Brown as a perspective student. During my studies in Brown, he continued to be an endless source of motivation and encouragement for me. Ugur taught me how to be a researcher, invested a lot of time teaching me how to effectively present my ideas and provided his valuable advice on many critical points in my career and personal life. I will always be grateful to him for his guidance, his support and his patience all of these years.

I would like also to thank my committee member and co-author, John Jannotti. His interesting questions and feedback helped me understand my own work in more depth and had an important impact on this dissertation. Thanks are also due to my third committee member, Stan Zdonik, whose support and encouragement helped me pursue my career goals.

I would also like to thank the student members of the Data Management Group in Brown. I am really happy I met and worked with all of you! Nesime, you have inspired me in many different ways, in my research, in my career choices (and even during our gym classes!). My dear officemates, Ying and Jeong-Hyon, thank you for sharing with me many stressful moments. Yanif, I really appreciate the effort and time you spent discussing the details of XPORT and building our demonstration. Finally, I would like to thank the summer interns who contributed to the prototyping of my work: Yenel Yildirim, Mohit Gogia and Swati Goyal.

The years I spent at Brown were the best years of my life thanks to my friends. Manos Renieris, Yiannis Tsochantaridis and Aris Anagnostopoulos patiently listened and advised me, especially in the early years. Ioanna Papaefthimiou and Ioanna Grypari were always bringing out the positive aspect of any situation and they made my life so much more fun. I would like also to thank Cleopatra Christoforou, Socratis Dimitriadis, Vaggelis Evaggelou, Dimitris Kazazis, Stelios Michalopoulos, Stefan Roth and Yiannis Vergados for all the laughs we shared. Finally, the last couple of years were much more enjoyable and less lonely than I ever expected thanks to Menia Pavlakou, Sofocles Mavroeidis, Babis Papamanthou and Dimitra Politi.

This thesis wouldn't have been a reality without the support of my family. First and foremost, I want to express my deepest gratitude to my husband, Nikos Triandopoulos. Nikos is the person who inspired me to apply to graduate school. He believed in me more than any other person in my life and he is always encouraging me to pursue more. My achievements wouldn't have been possible without his unconditional love and endless support.

Finally, I want to thank my parents, Grigoris and Evgenia Papaemmanouil. My parents made numerous sacrifices for my education and they helped me in any possible way during my studies. They encouraged me through all the difficult moments and they taught me to believe in myself. I want to thank them for allowing me to pursue my dreams, even when my decisions were very difficult for them. This thesis is dedicated to them.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The advent of the Internet as a global, easily accessible and fast evolving communication infrastructure has advanced network computing and enabled opportunities for building large-scale distributed applications. This new setting for developing applications with scale increased by several orders of magnitude led to systems with millions of nodes that seek to communicate and cooperate in a highly decentralized and heterogeneous environments. These systems consist of large numbers of geographically dispersed entities: sources that generate large volumes of data streams and consumers that register large numbers of interest subscriptions over these data streams, which are acquired, processed and then distributed to consumers in real-time.

In large-scale stream-based applications, data producers and consumers typically communicate through application-level networks (or *overlay networks*) consisting of nodes that cooperate with each other to route packets on behalf of any pair of communicating nodes. Using these overlay networks, endhosts are able to select by themselves paths with better performance and availability without relying on the underlying IP routing infrastructure.

Depending on their processing requirements, we divide stream-based applications into two types:

- *Dissemination* applications, which route streams from their producers to their consumers based on their subscriptions. These applications require stateless processing operations, which match the generated streams against the clients' subscriptions and identify the overlay routes to their interested parties. Example applications include multi-player online games, multicast-based content distribution, web feed dissemination [5], and stock ticker distribution.

- *Monitoring* (or *stream processing*) applications, on the other hand, require sophisticated processing of stream-based queries, e.g., complex aggregation and correlation over data streams. Examples are planetary-scale sensor network applications [4, 15], and network performance and security monitoring applications [2, 3].

Monitoring and dissemination applications are often developed using custom, ad-hoc approaches that hinder their implementation and maintainability. Furthermore, these systems are commonly optimized for a specific metric, which may limit their applicability to diverse applications and environments. Going forward, there is a need for general-purpose infrastructures that can effectively support a broad spectrum of these applications. This thesis introduces such an infrastructure, which is an extensible data stream acquisition, processing and dissemination system.

1.1 Design Goals

The design of this infrastructure was guided by three main goals:

- *Extensibility*: Monitoring and stream dissemination applications often exhibit diverse application logic and performance requirements. For example, a camera-based surveillance application may need to perform feature extraction over MPEG streams, whereas a feed-oriented application may involve XPATH queries over RSS streams. Moreover, different systems may have widely varying performance requirements and constraints. For example, a network intrusion detection application may have strict result latency requirements, whereas an environmental monitoring application running in a peer-to-peer setting may care more about fairness in bandwidth consumption. Hence, an extensible infrastructure is required that can be easily customized to support application-specific data types, stream processing logic and performance expectations.

- *Scalability*: Stream-based applications need to provide high network and workload scalability. The former refers to the ability to gracefully deal with increasing geographically distributed system components, whereas the later addresses large number of simultaneous stream producers and consumers. To achieve both types of scalability, the system should scale out and distribute its functionality across multiple nodes. Moreover, it should avoid global coordination as well as global knowledge of its nodes' state. Instead, its functionality should rely only on localized interactions and operations that utilize localized information.

- *Adaptivity*: Dissemination and monitoring applications are expected to operate over the public Internet, with large numbers of unreliable receptors, on commodity machines, some of which may contribute their resources only on a transient basis (e.g., in peer-to-peer settings). The highly dynamic nature of these environments requires the design of adaptive infrastructures, which can deal effectively to churn, time-varying workload, and resource availability.

In this thesis we introduce a distributed, adaptive infrastructure that can be easily customized for specific target applications and optimization metrics and can address the aforementioned design goals. In the next section, we present an overview of our approach.

1.2 Our approach

The designed of our infrastructure was motivated by the following observation: stream processing and dissemination applications have diverse logic and performance requirements, yet they all require several common facilities, which include construction, maintenance and optimization of an overlay network, routing and processing logic, and membership management. These applications are currently developed from scratch, requiring substantial effort and investment to "get it right" for each specific case. In contrast to existing approaches that provide point solutions to point problems, we introduce an application-agnostic infrastructure that can be easily customized for target applications. By abstracting over the common core functionalities of stream processing and dissemination systems, our solution achieves a clean separation between the "plumbing" and "application" and enables the system to uniformly support disparate dissemination and monitoring applications.

Extensibility is the central design consideration for our infrastructure, which supports diverse processing logic, stream and profile types, and optimization metrics. Specifically, we support 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 system. Both types of extensibility are implemented through a set of application-defined methods that encapsulate application-specific behavior and a cost model for defining desired performance metrics and constraints. Given these application-defined methods and performance expectations, the system automatically creates a data stream acquisition, processing and dissemination overlay network.

Our infrastructure relies on the publish-subscribe paradigm as its underlying communication model for routing streams from their producers to interested consumers, a design decision we argue has the potential to meet our objective for high *scalability*. Distributed publish-subscribe systems effectively decouple sources and destinations over geography and time: producers send data without knowing where and when the consumers will access them and consumers declaratively express their profiles and receive matching data without knowledge of specific data producers. This loosely-coupled architecture allows for high network scalability and robustness in the presence of churn, as these systems can gracefully deal with increasing geographically distributed sources and clients and dynamic changes in their membership.

To address the *adaptivity* requirement for our infrastructure, we designed a distributed, self-tuning optimization framework. This framework uses a novel, extensible, metric definition model that relies on the aggregation of cost metrics defined over the system entities (e.g., nodes, links, queries). Our model allows for the uniform specification of many commonly used performance measures, as well as new ones, through combinations of different aggregation functions and metrics. Moreover, the optimization process is driven by the iterative execution of metric-independent operations drawn from a set of *optimization rules*, which are designed towards system's convergence to a minimal cost configuration.

The optimization framework includes two types of optimization rules. *Network optimization rules* refine its structure by modifying the overlay connections among brokers. This set of rules includes pre-defined primitive rules as well as application-defined composite rules obtained through the composition of the primitive ones. We studied these rules in the context of publish-subscribe dissemination trees. More specifically, we used our infrastructure to build a peer-to-peer RSS feed dissemination service. We used this service 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 our system's flexibility and effectiveness, as it manages to improve each of these metrics significantly through its network-based optimizations.

The second type of optimization operations are the *query optimization rules*. We studied these rules in the context of monitoring applications that process continuous stream-based queries over the generated streams. The query optimization rules improve the network deployment of registered queries by migrating, replicating and partitioning query operators. We used our prototype to implement three distributions of feed-based processing queries, each one optimizing a different metric, including average query latency, maximum processing load across all nodes and total bandwidth consumption. Our experiments revealed that we can efficiently deploy stream processing queries over a network of brokers through a sequence of query distribution operations.

This thesis proposal report is outlined as follows. In Chapter 2, we provide general background on publish-subscribe systems and the stream processing model. Chapter 3 describes the basic models and concepts of our extensible infrastructure. Chapter 4 focuses on the network optimizations in the context of single-tree stream dissemination applications, while Chapter 5 concentrates on the query optimization operations that apply to monitoring applications. A detailed review of the related work is presented in Chapter 6. Finally, we conclude in Chapter 7, presenting interest directions for future work.

Chapter 2

Background

This chapter covers general background information on the concepts and models used throughout this thesis. In Section 2.1 we describe overlay-networks, which we utilized to construct the application-level networking infrastructure required by ISM applications. In Section 2.2 we discuss the publish-subscribe model, which is the communication model of our infrastructure. Finally, Section 2.3 covers basic concepts on streams processing, a data management and manipulation model used by stream-based applications.

2.1 Overlay networks

Overlay networks are virtual networks formed by cooperating nodes that share the underlying physical network. Similarly to a physical network, an overlay network has a topology consisting of the nodes of the network. However, the links between them are constructed by the agreement of the overlay nodes, creating a virtual topology. Nodes in an overlay are connected by virtual or logical links, each of which corresponds to a path, perhaps through many physical links, in the underlying network. Messages are transmitted only along these virtual links between the overlay nodes using the underlying unicast mechanism provided by IP. For example, many peer-to-peer networks are overlay networks because they run on top of the Internet. Dial-up Internet is an overlay upon the telephone network.

Overlay networks are suitable for routing messages to destinations not specified by an IP address. For example, distributed hash tables can be used to route messages to a node having specific logical address, whose IP address is not known in advance. In contrast to the Internet, in which routers are shared and thus, cannot be specialized for a particular purpose, the members of an overlay network may provide specialized services specific to the application at hand. An overlay-based multicast system can duplicate packets in the servers while a content distribution network can cache gigabytes of data. Examples of overlay networks include Akamai Technologies [1] for reliable, efficient content delivery, End System Multicast [33] and Overcast for multicast [35], and RON (Resilient Overlay Network) [10] for resilient routing, among others.

Overlay networks have also been proposed as a way to improve Internet routing. Previous proposals such as IP Multicast have not seen wide acceptance largely because they require modification of all routers in the network. On the other hand, an overlay network can be incrementally deployed on end-hosts running the overlay protocol software, without cooperation from ISPs. The overlay has no control over how packets are routed in the underlying network between two overlay nodes, but it can control, for example, the sequence of overlay nodes a message traverses before reaching its destination.

In general, overlay networks provide a flexible approach for applications to obtain new network semantics without modification of the underlying network. This characteristic makes overlays the most suitable model for designing a generic, networked infrastructure that supports a wide variety of ISM applications.

2.2 Publish-Subscribe systems

The publish-subscribe model [31] is an asynchronous, many-to-many communication model for distributed systems. It is an efficient way to disseminate data to a large number of clients depending on their interests. From a programmer's point of view, it is simple to use because it ensures that information is delivered to the right place at the right time.

The publish-subscribe model has data publishers that produce data and data subscribers that receive them. Subscribers describe the kind of data that they want to receive through their *profile* (or *subscription*). Data produced by the publishers will subsequently be delivered to all interested subscribers with matching interests.

A publish-subscribe system implements the publish-subscribe model and provides a service to applications by storing and managing profiles and asynchronously disseminating events. An important feature of the publish-subscribe model is that clients are *decoupled*. A publisher does not need to know all subscribers that receive the data it produces, and similarly, subscribers do not know the identity of publishers that send data to them. All communication between them is handled by the publish-subscribe system. Thus, this loose coupling removes dependencies between clients so that the publish-subscribe model is a *scalable* communication paradigm for large-scale systems.

Many publish-subscribe systems exist that implement different variants of the publish-subscribe model. A major distinction is how profiles express the interest of subscribers. Two main forms of the publish-subscribe model have resulted from this, topic-based and content-based publish-subscribe. In topic-based publish-subscribe, publishers publish data with respect to a topic or subject. Data subscribers specify their interest in a topic and receive all data messages published on this topic. Therefore, topics can be seen as groups in group communication. This makes the implementation of a topic-based publish-subscribe system simple and efficient since it can be built on top of a group communication mechanism such as IP multicast.

However, ISM application like network monitoring or feed-based dissemination would benefit from a more precise specification of interests than just a topic name. Subdividing the event space into topics has the disadvantage that it is inflexible and may lead to subscribers having to filter data coming from general topics. To address this limitations content-based publish-subscribe systems have emerged. In content-based systems profiles can be any function over the data content. This provides higher degree of flexibility and expressiveness, making this model suitable for designing generic publish-subscribe systems. Hence, in the rest of the section we concentrate on content-based publish-subscribe systems.

Content-based publish-subscribe In a content-based publish-subscribe system, the structure of a profile is not restricted — it can be any function over the content of the data message. This introduces a trade-off between scalability and expressiveness [17]. The more expressive a profile becomes, the more difficult it is to evaluate it, increasing the overhead in the publish-subscribe system. A content-based profile usually depends on the structure of the data. The structure can be binary data, name/value pairs, or even semi-structured data such as XML. A profile is expressed in a language that usually specifies a filter expression over data messages.

Depending on the application's data types and the complexity of its profiles, publish-subscribe dissemination systems may use their own algorithms and *indexing structures* for efficiently storing profiles and matching incoming data against them. ONYX [30] uses YFilter [29] and XRoute [25] employs the XTrie index [24] for matching XPath profiles, whereas SIENA [18] uses a custom index [20] for storing and matching relational profiles.

In centralized implementations of a publish-subscribe system, all subscribers forward their profiles in a single server. This server also receives all generated data from the registered publishers and is responsible for matching incoming data messages to the specified profiles and identifying the subscribers each data message should be forwarded to. The main drawback of this approach is that the data matching and forwarding cost is allocated to a single broker, limiting the scalability of the system. Thus, distributed publish-subscribed systems have been proposed, aiming to distributed this load to a set of nodes and address the scalability concerns.

The dissemination infrastructure of a distributed publish-subscribe system consists of a set of nodes (also referred as *brokers*) organized into an overlay network. Publishers and subscribers are clients that first need to connect to a broker in the logical overlay network. Data messages published by the publishers are then routed through the overlay network of brokers depending on the profiles submitted by subscribers. The process of *content-based routing* [19] can be viewed as a distributed implementation of a data matching algorithm so that events are delivered to all interested subscribers.

Although several topologies for the network of event brokers are proposed, we will only consider the general peer-to-peer topology because it is the most realistic for a large-scale network. On top of this general network topology, brokers are usually organized into one or more application-level dissemination trees [18, 30, 44]. These trees are created based on the idea of reverse path forwarding of messages in broadcast algorithms [28]. More specifically, a subscriber registers its interest by forwarding its profile to a broker. This profile creates state in the overlay network by being propagated upstream to the root of the tree. When profiles propagate, they create entries in routing tables that store (1) the profile, (2) the last hop, and (3) the next hop for this message. Thus, profiles create reverse routing paths from the root of the tree to the subscriber. 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.

It is obvious that this communication model can satisfy the scalability and adaptivity to churn requirements of ISM applications. A problem is that a pure publish/subscribe system is of limited use to distributed dissemination applications designers because the implementation of the communication model will be specific to each applications data types and optimization metrics. A better solution is to provide an extensible publish-subscribe system that complements the scalable and efficient implementation of the publish-subscribe model with additional functionality.

2.3 Stream-based processing

Data Stream Processing systems (DPSs) [16, 26, 27, 42] has emerged in response to applications that must deal with stream, that is possibly unbounded sequences of data elements. These data streams are typically generated rapidly one after the other, and usually require real-time processing.

Common stream sources include sensors for collection of physical measurements about their environment (e.g. a temperature sensor), and software programs that report the occurrence of certain events of interest (e.g. a program reporting the trades of stock shares). An important property of these data sources is that they are push-based, i.e., they are not programmed to store and provide data on demand, but to release it as soon as new data becomes available. There are an increasing number of applications that require continuous monitoring and processing on data streams. Examples include sensor-based monitoring (e.g. biomedical monitoring, road traffic monitoring), financial analysis applications, GPS-based location tracking, and network traffic monitoring.

A DSMS provides the same kind of infrastructure to stream-based applications that Database Management Systems (DBMS) have provided for data processing applications. In traditional data processing systems, large volumes datasets are persistently stored on disk, and one-time queries are executed on them. In this model, data is pulled from the disk as it is demanded by the queries. In case of data streams, these roles are completely reversed. A large number of long-lived queries called continuous queries are defined in advance of the data, and are persistently stored in the system. Data streams, as they arrive, are evaluated through these standing queries. In this model, data is pushed from the sources and needs to be processed through the queries.

Data streams are possibly infinite sequences of data elements, referred to as *data tuples*. Tuples are composed of attribute values and all the tuples contained in the same stream have the same set of attributes. The set of attributes of a stream is called the *schema* of the stream.

In a stream-based processing model queries can be described by a dataflow diagram. Each box represents a query operator and each arc represents a data flow or a queue between the operators. Stream processing engines [16] process a large number of queries that are built out of a set of operators. Each such query has an arbitrary number of input streams and a single output. An operator may be connected to multiple downstream operators. All such splits carry identical tuples and enable sharing of computation among different queries. Multiple streams can also be merged by the operators that accept more than one input. A query network is a collection of such queries.

Stream-based operators process tuples and producing results of interest. Depending on the state they require to perform their processing, we distinguish two types of stream processing operators, *stateless* and *stateful*. In this section, we briefly discuss both types of operators.

Stateless Operators Stateless operators perform their computation on one tuple at a time without holding any state. Here, state refers to the data structures that an operator maintains to perform its operation. Examples of stateless operators include: *filter*, *map*, and *union* operations. Filter is the equivalent of a relational selection operator. It applies a predicate to every input tuple, and forwards the ones that satisfy the predicate to its output stream. Tuples that do not satisfy the predicate are either dropped or forwarded to a second output stream. A map operator extends the projection operator in relational algebra: it transforms input tuples into output tuples by applying a set of functions on the tuple attributes. Finally, a union operator simply merges a set of input streams (all with the same schema) into a single output stream. Union merges tuples as they arrive without enforcing any order on the output tuples.

Stateful Operators In contrast to stateless operators, stateful operators produce output tuple as a result of computation over a collection of input tuples. Here, we present one representative stateful operator, namely *aggregate*. An aggregate operator computes an aggregate function such as average, maximum, or count. The function is computed over the values of one attribute of the input tuples. Before applying the function, the aggregate operator may partition the input stream based on the values of one or more other attributes (e.g., produce the average temperature for each room). To deal with the infinite nature of input streams, aggregates conduct computations over windows of tuples (e.g., produce the average temperature every minute). These windows are defined over the values of one attribute of the input tuples, such as the timestamps of tuples. Windows can also be defined in terms of the number of tuples (e.g., windows of ten tuples can be defined for every two tuples). Both the window size and the amount by which the window slides can be parameterized.

Chapter 3

Basic Models and Concepts

In this chapter, we describe the basic concepts and the core functionality of our infrastructure. These concepts form the basic building blocks for the extensible dissemination and stream processing systems we built. We introduce the underlying communication model in Section 3.1. We concentrate on its extensibility aspects and describe the general metric definition model in Section 3.2. Finally, we outline our optimization framework in Section 3.4.

3.1 Overlay construction

Our infrastructure relies on the publish-subscribe paradigm as its underlying communication model. More specifically, we rely on the content-based data dissemination model, where data producers publish their generated data and data receivers express the content of the data they wish to receive. As mentioned in Section 2.2, this model provides many advantages, namely source and client decoupling, scalability and adaptivity to subscription changes. In order to take advantages of these properties, we disseminate *all* generated streams through content-based publish-subscribe trees. Our infrastructure is responsible for:

- constructing the overlay content-based publish-subscribe trees connecting the publishers with the participating subscribers and
- routing the generated data, based on their content, only to the interested consumers (referred to as *content-based routing*).

In this section, we describe how a single dissemination tree is created, given a set of brokers, sources that generate streams and subscribers which specify the streams of interests through their profiles.

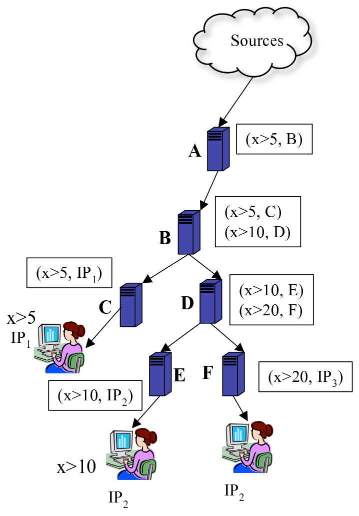


Figure 3.1: Construction of profile-driven publish-subscribe tree.

Constructing an overlay publish-subscribe tree The basic application-level overlay tree of our infrastructure consists of a subset of the available brokers. In order to join the tree, nodes select an existing node as its parent. Generally, 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 better network location. Hence, nodes pick a random node from the available ones as their parent. However, this parent selection process should not violate any specified constraints, if any defined by the application (e.g., node fanout).

In certain cases we modify our general approach and initially construct a *best-effort* overlay tree. Under this approach nodes collect information about the available nodes and connect to the "best" node, as defined by the application's criteria. For instance, if the application aims to reduce dissemination latency among nodes, then, the joining node would ping the available ones and collect their latency from the root of the tree and their latency from the new node. Based on these latency values, it connects to the one with the minimum total latency. Generally, it connects to the node that adds the minimum overhead on the application's optimization metric.

Once a node has decided on its parent, it connects to that node and registers its profile. The node adds the new profile along with the address of the new node as a new routing entry. Based on this entry, it can route to the new node all incoming data that match the node's (or client's) profile. An example is shown in Figure 3.1. Leaf nodes maintain routing entries to their clients along with their profile, while intermediate nodes maintaining routing entries for their children in the tree. Every new profile is disseminated towards the root of the tree and on every intermediate node it is merged with the existing profiles (if any). Profile merging aims to reduce space and processing requirements on the intermediate nodes by storing only the more general profiles expressing their subtree's interest. This general profile will "cover" any profiles registered in the node's subtree. For example, in Figure 3.1, node D, propagates to its parent, node B, the profile $x > 10$, as this covers the profiles of its subtree. If a new general profile is produced by the merging process, then the node will propagate it up the tree towards the root, until it reaches either a node with a more general "covering" profile or eventually the root. This process leaves the root of the tree with a profile that covers all registered profiles.

The root of the tree is responsible for acquiring the streams either by pulling the corresponding sources or by providing a pushed-based interface that handles and forwards any incoming data streams. Upon the receipt of a data message, a broker checks all the entries in its routing table to determine whether the message should be forwarded to a downstream broker (or a client).

Based on this single-tree communication model, we defined our basic extensible cost model and our optimization framework, which are described in the following sections.

3.2 Extensibility

The central design consideration for our infrastructure is extensibility: it supports an extensible set of data and profiles/query types, optimization metrics and constraints. For the rest of the discussion, we will use the terms query and profile interchangeably. We support two types of extensibility:

- *Profile-related extensibility* allows for easy specification and integration of 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 system.

Given application-defined data types, profile types and performance metrics, the system automatically builds, maintains and optimizes an overlay network consisting of the available broker machines in the system. Depending of the application, brokers are customized to perform (or not) different types of stream processing, however, in all cases, the overlay network is responsible for routing generated streams from their producers to the interested consumers.

While profile-related extensibility depends on the processing requirements of an application (i.e., simple content-based routing or advanced stream processing), cost-related extensibility is based on a common metric definition model. Hence, in the rest of the section we concentrate on our approach for supporting cost-related extensibility and describe our generic metric definition model. We will refer to profile related extensibility in Chapter 4.

3.3 Generic Metric Definition

Our basic metric definition model is based on a novel *aggregation* schema that defines the performance expectations of an application. The model allows applications designers to express cost metrics for the global system cost, e.g., average bandwidth bottleneck, maximum dissemination latency, as well as cost metrics for each broker, e.g., processing load, outgoing bandwidth consumption.

Our metric definition model specifies performance criteria for a single publish-subscribe dissemination tree. Thus, for the rest of the discussion, we assume that each node has a single parent and a set of children in this tree. In Chapter 5, we will describe how this model is extended for multiple dissemination trees.

In our model, the *system cost* is defined through a *two-level* aggregation model. In the first level, it computes the cost of each node (*node cost*) by aggregating application-defined statistics collected from the node's local neighborhood. The second level defines the cost of the system by aggregating the node costs. Both aggregations are defined by the following general signature:

$$\boxed{\text{aggregate (aggregation function, value, aggregation set)}}$$

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 statistics. Similarly, applications can also specify constraints for each node. Figure 3.2 shows the general grammar for defining the performance criteria. We now describe our grammar and the two-level aggregation model in a bottom-up manner.

Local statistics

To facilitate the expression of a broad spectrum of metrics, nodes maintain some built-in performance *local statistics* like overlay link latency, incoming data rate, etc. Moreover, they maintain state related to their local profiles and that of their neighbors, e.g., the client profiles, its children's profiles. Applications define their own *local metrics* using either use these built-in statistics or computed them by applying an application-defined arbitrary function over a set of the local statistics. For example, the incoming data rate can be defined based on the selectivity of the node's profiles.

A local metric may refer 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, respectively, in the metric's implementation method. The exact links on which the local metric will be calculated is determined by the *node cost set* term of the grammar. If this term is set to PATH,

01. system cost:= aggregate (aggregation function, node cost, system cost set)
02. node cost:= aggregate (aggregation function, local metric, node cost set)
03. node cost set:= PATH|CHILDREN
04. local metric:= f(local stats, LINK|NODE, node cost set)
f(LINK|NODE, node cost set)
05. local stats:= path latency|incoming data rate|local profiles...
06. constrained metric:= (system cost|node cost, operator, threshold)
07. operator:= $<$ | $>$ | $\leq$ | $\geq$ | $\neq$

Figure 3.2: General metric definition grammar

then the local metric is measured on the link to the node's parent, while if it is set to CHILDREN the local metric is measured on the links to the children. This is described in the following section.

Node cost

The node cost can be defined as an aggregation of the local metrics of some neighboring nodes:

aggregate (aggregation function, local metric, PATH | CHILDREN)

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 that are frequently used 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.

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 brokers in the tree:

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

The *system cost set* refers to the set of cost metrics we will aggregate. In this case, this set is the total set of nodes participating in the dissemination tree.

Constraints

Often, application designers wish to express constraints on certain attributes of the network. For example, they may need to limit the fanout of the nodes (i.e., the number of their children) in order to reduce the outgoing bandwidth requirements of the nodes. Our model allows constraints to be expressed based on the following generic signature:

constraint (metric, operator, threshold)

Constrained metrics can be defined in the same way as the system cost or the node cost, i.e., following the two-level aggregation model. An operator and a threshold for the constrained metric should also be specified. 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. Our system customizes its functionality and optimization framework to respect these constraints. This process will be described in Chapter 4 and Chapter 5.

3.4 General optimization framework

Our infrastructure strives to meet three requirements. First, it must be *scalable* in terms of the number of brokers and subscriptions (or queries) in our network. Second, it should be *efficient* and *adaptive* to time-varying network or workload conditions. Finally, it should be *extensible* with respect to the performance measures it can support *and* optimize. To address these challenges, we designed a general, decentralized optimization framework that strives to create an overlay network and identify in-network query deployment that minimizes application-specific cost functions. One unique characteristic is that our framework is *metric-independent*; it uniformly applies to and handles a variety of cost metrics.

A brute-force approach for identifying the best network configuration or the best query in-network deployment would be to consider all possible reconfigurations of our system. 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 possible optimization may be prohibitively expensive. Instead of performing an exhaustive search of all configurations, XPORT limits its search to a smaller set of "promising" reconfigurations, called *optimization rules*. The optimization process is driven by the iterative execution these metric-independent rules, which are designed towards the system's convergence to a minimal cost configuration.

Our approach does not rely on global information and has low communication overhead. Each node maintains information about its own "neighborhood" and applies local optimization rules 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"). Periodically, we consider specific non-localized optimizations that help our system avoid local optimum. Our framework exploits its structured cost-definition model and the semantics of the aggregation functions to derive generic properties and equations to quantify the benefits of any candidate optimization operation. Our cost-model for evaluating the effect of an optimization operation depends on the type of optimization rules (network or query optimization rules) and so will be described in detail in Chapter 4 and Chapter 5.

In the next sections, we provide a high level outline of the basic optimization approach followed by our infrastructure. In general, our optimization framework addresses the following challenges:

1. What type of optimization rules are require?
2. How can we efficiently quantify the impact of an optimization rule on the system cost?
3. How we decide which optimization rule should be applied?

3.4.1 Optimization rules

Optimization rules are categorized in two types, depending on their scope, i.e., the set of nodes they take under consideration and hence, the set of node they could affect:

- *Local optimizations*: Nodes consider modifications of their local neighborhood, e.g., parent, children, grandchildren, etc.
- *Directed optimizations*: Nodes consider non-localized network areas that could potentially yield higher performance improvement than the local neighborhood.

Local optimizations We informally define a *local optimization rule* as one that requires interactions among only the "nearby" brokers on the overlay tree. This set of "neighbors" of a broker is defined by the *optimization unit*. For the basic publish-subscribe dissemination tree of our communication model this unit is shown in Figure 3.3. This includes a broker n_i, its parent n_p and its children. We refer to these nodes collectively as the *optimization unit* of n_i. Local optimization rules can change either the structure of the network, or, in the case of stream processing applications, modify the deployment and execution of queries across the network brokers. These rules are defined based on the scope of the optimization units. Moreover, they can be statically or dynamically composed to define complex but more powerful composite optimization operations.

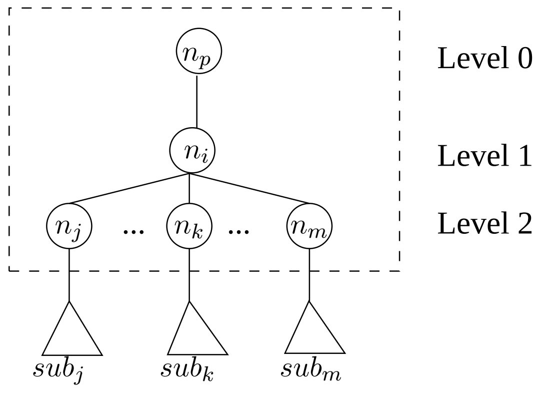


Figure 3.3: Optimization unit of n_i.

Figure 3.3 shows the minimum optimization unit in our system. However, optimization units can be defined differently; in particular, they can be extended to include more nodes in the network. Extending the optimization scope in this manner increases the flexibility of the framework, as it facilitates the definition of a larger number of more powerful rules. On the downside, such an extension also increases the maintenance traffic and the size of the state nodes need to collect and maintain in order to apply optimization rules in this unit. We will refer in detail on this trade-off in Chapter 4.

The main advantage of relying on localized optimizations is that it reduces the search space of candidate optimizations. In the case of network optimization rules, each node will considering restructuring nodes in its local neighborhood. Moreover, if we attempt to optimize stream processing queries, e.g., through operator migration, our localized approach will limit the space of candidate locations only to the nodes in the neighborhood of the operator's current location. Furthermore, non overlapping neighborhoods can be optimized locally and concurrently, allowing the system to converge faster in a lower-cost configuration.

Directed optimizations Although our experimental results showed that local optimizations can yield significant improvements for some metrics, we also discovered that certain measures are more prone to local optimum. Our system selectively disseminates network statistics, on the basis of the optimization metric, that allow nodes to efficiently identify promising non-local operations. These operations focus on specific low cost network areas and we refer to them as *directed optimizations.*

As a result of the statistics collection process, our nodes maintain certain local and aggregated cost metrics. Our infrastructure selectively disseminates these metrics across nodes, allowing the discovery of alternative neighborhoods that could be utilized and improve the system performance. We rely on the definition of the cost metrics, specifically on the semantics of the aggregation functions, in order to determine: (i) which nodes may be interested in the statistics, and (ii) which statistics could be of potential interest. Moreover, we utilize the dissemination trees to create filter-based routing paths that forward statistics from their producers only to the interested consumers. Nodes can exploit these statistics in order to identify promising optimization areas.

3.4.2 Impact evaluation of an optimization rule

The goal of the local and directed optimizations is to improve the overall system cost. Our framework calculates the benefit of every candidate rule and applies the best one. Since the system cost is an aggregation of specific node cost metrics, the exhaustive approach for quantifying this cost benefit is to estimate the value of these node cost metrics after the optimization and aggregate them to get the new system cost. This approach may have prohibitively high communication overhead. However, our framework avoids this overhead as it understands the semantics of the aggregation functions and the definitions of the cost metrics. This knowledge allow us to efficiently quantify the impact of an optimization rule.

More specifically, quantifying the benefit of an optimization rule relies on the notion of *dependencies*. We identify cost dependencies across all entities over which the system cost is defined, i.e., we define which nodes have an impact on their peers cost metric. This can be derived from the definition of our cost metrics. For example, if a node aggregates the local metrics of its children to evaluate its own cost metric, then, any change on its children metrics will project on its own cost as well. Thus, for every optimization rule we quantify the impact on the affected entities. Furthermore, based on the semantics of the aggregation functions, we can automatically identify the state required by each node, as well as the information to be exchanged among nodes during the optimization. We maintain state regarding depended entities and, whenever feasible, we aggregate this state to reduce the optimization traffic and state size.

3.4.3 Bottleneck-based optimization

Our framework uses a *bottleneck-based* approach for optimization in which it focuses only on *effective* optimization rules that have the potential to reduce the overall system cost. Other rules, 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 we will focus solely on the most loaded node and attempt to decrease its cost. To implement this approach, we rely on the notion of *critical entities*. For example, a node is considered critical if a change in its cost may potentially affect the system performance.

Our optimization proceeds over *optimization periods*. At each period, each node that can affect a critical entity exchanges data with the nodes in its unit, and quantifies the benefit of the candidate optimization rules. It then identifies the most effective rule and sends it to the root of the tree. The root simply decides on the best rule (i.e., with the maximum expected benefit) and informs the corresponding to 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 rule is identified.

Chapter 4

Network Optimizations for Dissemination Applications

In this chapter, we introduce XPORT (eXtensible Profile-driven Overlay Routing Trees), a generic publish-subscribe distributed data dissemination system. XPORT uses our basic communication model and creates an overlay content-based publish-subscribe tree that disseminates streams from their producers to any interested subscribers. Our system provides the core dissemination infrastructure for a growing set of dissemination-based applications and services, including web feed dissemination (RSS/Atom), multicast-based content distribution, multiplayer network games, stock ticker distribution and large-scale distributed collaborative applications.

By abstracting over the core functionalities of publish-subscribed dissemination systems, XPORT supports *profile-related extensibility*: it can be customized for a wide variety of dissemination applications through a small set of methods that encapsulate application logic. It also builds on top of our general extensible cost model and supports *cost extensibility*: it can express of a large set of performance measures and constraints, including ones that involve multiple metrics.

XPORT's optimization framework is driven by a set of rules that continuously refine the *structure* of its dissemination tree. We refer to these rules as *network optimizations*. The set of optimization rules is extensible: applications can define their own *composite* optimizations by combining a set of *primitive* rules provided by XPORT. XPORT relies both on localized and directed optimization rules, as they were defined in Chapter 3. Based on these operations, it consistently delivers efficient network configurations with near-optimal performance. Our framework is supported by a set of statistics dissemination and approximation techniques that maintain low network traffic.

We introduce the system's API in Section 4.1 and its optimization in Section 4.3. We describe the basic XPORT architecture and its run-time behavior in Section 4.4 and present our experiments and results in Section 4.6.

4.1 XPORT API

Based on the main functionality of data dissemination systems, we identified the types of methods an application needs to define in XPORT, in order to support an extensible set of profile types. 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. Thus, each application designer should provide the implementation of an interface that includes the following two methods:

- **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.

Our implementation is based on dynamic method invocation using the reflection package available in Java. More specifically, our system is designed to define in run-time which method to invoke, given the name of the class implementing the above interface. Thus, application designers simply need to implement the above interface and provide the name of the implementation class as a parameter to our system. Moreover, each application designer should provide the implementation for the following classes:

```
01. system cost:= aggregate (system cost function, node cost, system cost set)
02. system cost function:= MIN|MAX|SUM|AVERAGE|PRODUCT|VARIANCE|STD
03. system cost set:= BROKERS|CLIENT|BROKERS—CLIENTS
04. node cost:= aggregate (node cost function, local metric, node cost set)|
                local metric| g(node cost)
05. node cost set:= PATH|CHILDREN
06. node cost function:= MIN|MAX|SUM|AVERAGE
07. local metric:= f(local stats, LINK|NODE, node cost set)
                f(LINK|NODE, node cost set)
08. local stats:= path latency|incoming data rate|local profiles...
09. constrained metric:=(system cost|node cost, operator, threshold)
10. operator:= < | > | ≤ | ≥ | ≠
```

Figure 4.1: XPORT's grammar

- **Profile class.** The class defines the type of profiles the application will use.
- **Root class.** The root class defines the functionality for the root of tree. It is used to specify the push/pull-based interface between the root broker and the sources. In the case of a pull-based system, the root class specifies how to fetch data messages from the source and forward them to XPORT. For the push-based systems, it specifies how to handle incoming messages from the sources.

4.2 Cost definition model

In order to support cost-related extensibility, XPORT implements and extends the general cost model defined in Chapter 3. In particular, it implements specific aggregation functions and extends the definition of the node cost to allow the definition of multi-metric overlays. In this section, we describe these extensions in detail.

Aggregation functions. In order to generalize the aggregation technique and make the presentation more succinct, we categorize the aggregation functions used by XPORT into three classes:

- *additive functions*: SUM, AVERAGE,
- *bottleneck functions*: MIN, MAX and
- *holistic functions*: VARIANCE, STANDARD DEVIATION STD, 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

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 4.1: Two-level aggregation examples.

on the cost 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.

Figure 4.1 shows the extended grammar used in XPORT. The system cost can use all types of aggregation functions, while the node cost uses only the additive and bottleneck functions. Table 4.1 shows the different aggregation function combinations along with some example metrics.

Node cost. We generalized further the definition of the node cost which, in XPORT, can be defined as (i) an application-defined local metric, (ii) a combination of metrics defined as the node cost, or (iii) an aggregation of the local metrics of some neighboring nodes (as described in Chapter 3). Based on this extension, XPORT can create multi-metric overlay networks by allowing applications to specify the node cost as a combination of multiple optimization measures, e.g., the product of latency and bandwidth.

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 needs to forward it to any interested descendants.

```
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 × bandwidth
path latency = aggregate (SUM, link latency, PATH)
bandwidth = aggregate (MIN, link bandwidth, PATH)
```

4.3 Distributed optimization

As mentioned in Chapter 3, the goal of the optimization process is to minimize the application-specified system cost and, through a set of optimization rules, adapt to time-varying network or workload conditions. In XPORT these rules modify the structure of its tree. We refer to these rules as *network optimization rules*. In the rest of section we describe in detail this rules. We start from the local optimizations and continue with our set of directed optimizations.

4.3.1 Local network optimizations

To allow the definition of powerful local rules, we extended the minimum optimization unit described in Chapter 3. XPORT's unit consist of brokers that are at most three levels from each other; it includes a broker n_i, its parent n_p, its children *and* its grandchildren, as shown in Figure 4.2.

XPORT's local optimization rules are *transparent* outside its optimization unit; i.e., the rule 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

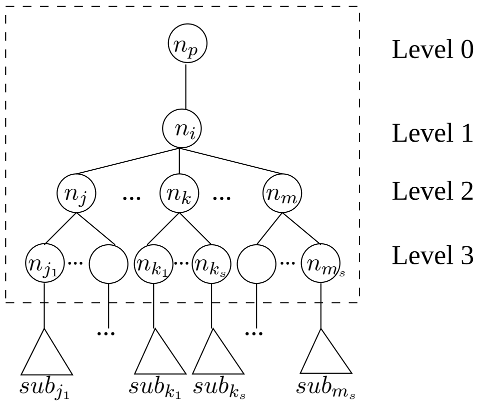


Figure 4.2: Optimization unit of n_i in XPORT.

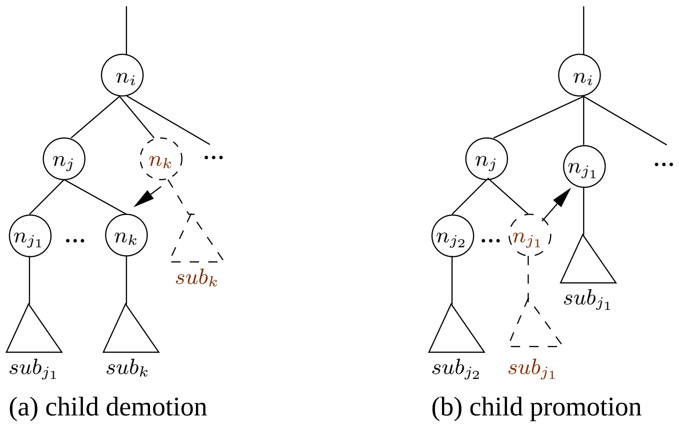
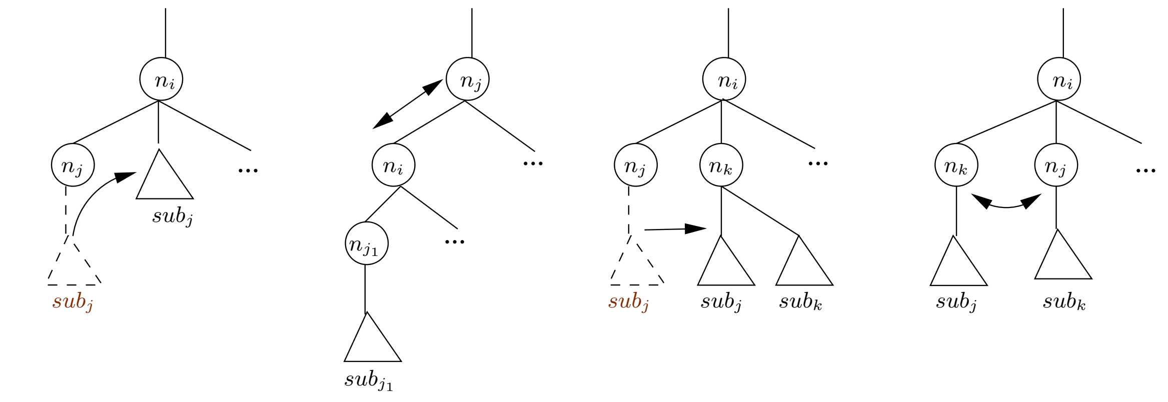


Figure 4.3: Primitive local optimization rules

last level will continue forwarding the same data to the same nodes, as they did before applying the optimization rule. 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 an optimization rule. Keeping the topological and profile-related effects of our optimizations local reduces the cost of network reconfiguration, as fewer nodes are affected by each rule.

XPORT's localized optimizations contains a number of built-in *primitive optimization rules* as well as other composite operations defined by the application. Our experimental results show that a small number of well-chosen rules can be very effective while incurring low overhead.



(a) subtree promotion (b) parent-child swap (c) subtree migration (d) sibling swap

Figure 4.4: Composite local optimization rules.

Primitive optimization rules

XPORT's primitive rules are *child demotion* and *child promotion* (Figure 4.3(a) and (b)). We explain these optimizations with respect to the optimization unit in Figure 4.2.

- **Child demotion.** This rule 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 rule 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.

Composite optimization rules

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

- **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 4.4 (a)). This rule can be derived by applying the promote child operation to every child n_{j_s} of n_j.
- **Parent-child swap.** In this rule, the owner of the optimization unit n_i and its child n_j swap positions without moving the subtree of n_j (Figure 4.4(b)). This will force every subtree

Optimization Rule	**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 4.2: Primitive and composite local optimization rules.

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 4.4(c)). 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 rules, 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 4.4(d). 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 4.2 shows how these optimizations rules are created from the primitive ones[1]. Similar tree optimizations were also used by previous work [14, 55].

4.3.2 Directed optimization rules

Our experimental results showed that localized optimizations are very powerful for specific metrics, allowing XPORT to converge very close to optimal network configurations. Specifically, most additive metrics we implemented (e.g., average latency, bandwidth consumption) where optimized using only local optimization rules. However, we discovered that bottleneck metrics were more prone to local optimum. To avoid local optimum and efficiently optimize bottleneck metrics, we allowed

[1](In Table 4.2 '$\rightarrow$' 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).

XPORT to apply a specific set of non-localized optimization rules, called *directed optimization rules*. These rules involve nodes from different optimization units and they aim to re-utilize network areas with good performance. XFlow utilizes the available network statistics and based on them nodes can identify these promising non-local operations. In this section, we describe our approach.

Statistics management XPORT nodes maintain certain local and aggregated statistics which they *selectively* disseminate to discover alternative neighborhoods that could improve the performance. XPORT relies on the definition of the system cost metrics, i.e., on the semantics of the aggregation functions (i.e., MAX, SUM, etc), in order to determine: (i) which nodes may be interested in the statistics, and (ii) which statistics could be of potential interest. Moreover, it creates filter-based routing paths on top of the existing overlay tree that forward the useful statistics from their producers only to the interested consumers. The statistics propagation process runs with a frequency that reflects the workload and network changes of the specific application.

Statistics selection Each node stores a local metric and its cost metric. It periodically disseminates these statistics and informs its peers about the performance of its neighborhood (i.e., the path to the root or its children). Specifically, the local metric reveals the node's resource utilization (e.g., processing load) or its immediate neighbors' properties (e.g., latency to its parent), while the aggregated node cost provides a performance measure of the path leading to the node (or of the links to its children). Table 4.3 shows which value each node will forward (*Disseminated Statistics* column), which dependents only on the aggregation function used for the definition of the node cost. The table shows that each node forwards statistics of constant size ($O(1)$) and this size is independent of the number of profiles as well as the number of nodes.

Selective dissemination Nodes are interested only on network components (links, paths, nodes) that perform better than their own local neighborhood. Table 4.3 shows the predicates used by each node n_i to filter out non-informative statistics (*Filters* column). XPORT uses the aggregation semantics to *automatically* customize these filters. Additive functions imply that the node cost will be higher than the local metric, thus the lower-bound filter is their local metric. For example, for the node latency case, n_i receives statistics about paths with less latency than its latency to its parent (i.e., its local metric). In the case of the MIN function, the node cost provides the lowest value n_i is interested in; n_i will receive statistics about links and paths with higher bandwidth than the capacity of its own path to the root.

Furthermore, XPORT distributes statistics only to nodes that may affect the global system cost. Table 4.3 shows the conditions that should hold in order for a node to receive any statistics from its peers. These conditions are *agnostic* of the actual optimization metric and depend on the aggregation

Node Aggregation	**Disseminated Statistics**	**Filters**	**System Aggregation**	**Condition**
SUM	node cost	<node cost	SUM	
			MIN	node cost= system cost $\pm\delta$
MIN	local metric node cost	>node cost	SUM	
			MIN	node cost = system cost $\pm\delta$

Table 4.3: Statistics filtering and dissemination per node.

functions that evaluate the node and system cost. For example, if the optimization metric is the average node latency, then every node can improve the system cost by reducing its own latency. On the other hand, in order to minimize the maximum node latency, we would forward statistics only to the node with the highest dissemination latency c, (or with small difference $\pm\delta$ from the worst response time).

Statistics routing XPORT uses the structure of its tree to distribute its statistics. It avoids flooding the network by constructing predicate-based routing paths that filter out unwanted statistics as early as possible. For example, we avoid forwarding load information of the most loaded node. To construct these filtering paths, nodes propagate their statistics filters (Table 4.3) to their parent. Each node aggregates the filters of its children in each tree and propagates the aggregated filter upstream towards the root. This allows nodes to be aware of the interests of their descendants and selectively route filtered statistics to the interested nodes.

Directed operations

XPORT considers a set of specific directed operations that allows nodes (or subtrees) to be migrate to different optimization units, as well nodes to swap their positions in the dissemination tree. Specifically:

- **Child move.** This rule moves n_i along with its subtree under new parent n_k outside n_i's optimization unit.
- **Subtree move.** This rules moves all children of a node n_i under a new parent n_k in a different optimization unit.
- **Node swap.** This rule swaps two nodes n_i and n_j in different optimization units. Node n_i becomes the parent of n_j's children and vice versa for the n_j.

4.3.3 Evaluating global cost changes

Our optimization framework uses a tree-oriented cost model to estimate the benefit of each potential optimization rule. With the knowledge of the semantics of the aggregation functions and optimization rules, the system derives and automatically collects the required state and statistics. In this section, we describe the details of our approach.

We start our section by providing the general equations that estimate the cost benefit of an optimization rule. We start with some definitions and continue with our metric-independent equations for the cost benefit.

Definition 1. *Let $cost_i$ be 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 in the subtree of 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, in this case, the node cost is defined as a local metric, XPORT limits, whenever possible, the dependence set based on the definition of this metric. For example, if the local metric 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 4.2). 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 n_i's unit.* Intuitively,

Symbol	Definition
$cost_i$	cost metric of n_i
D_i	dependents of n_i
$cost(D_i)$	dependence set cost of n_i
U_i	nodes in optimization unit of n_i
S_i	nodes in the last level of the optimization unit of n_i
L_i	dependents of the optimization unit of n_i
b_i	benefit of an optimization rule in the optimization unit of n_i
$children_i$	children of n_i in the dissemination tree
c	current system cost
g_i	minimum local metric for n_i
$h_i(j)$	cost of dependent n_j relative to n_i

Table 4.4: Model Terminology

this is the set of nodes that do not belong in the optimization unit of n_i but may be affected by any optimization rule applied on its unit.

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

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

$$b_i = \sum_{j \in U_i} (cost'_j - cost_j) + \sum_{k \in S_i} \Delta cost(D_k) \tag{4.1}$$

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 optimization rule. However, depending on the rule, many of its terms are zero, so simpler equations can be obtained. Examples of these equations are given in Table 4.5.

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 4.5: Simplified cost equations for local optimizations and an additive (SUM) system cost function.

Bottleneck functions We assume that the system cost function is MIN. Then n_i estimates the new minimum cost across 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 optimization rule. If c denotes the current system cost, then the benefit of the rule is:

$$b_i = c - \min_{j \in U_i, k \in S_i} \{cost'_j, cost'(D_k), \min_{m \notin U_i, m \notin L_i} \{cost_m\}\} \tag{4.2}$$

where $cost'(D_k) = cost(D_k) + \Delta cost(D_k)$. To estimate the minimum cost of all nodes not affected by the optimization rule state of constant size is required at every node. We will provide details in the Section 4.3.3.

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

$$\begin{aligned} b_i &= c - \frac{1}{|V|-1} \sum_{j \in V} (cost'_j - \overline{cost'})^2 \\ &= c - \frac{1}{|V|-1} \{ \sum_{j \in U_i} (cost'_j - \overline{cost'})^2 + \\ & \sum_{j \in L_i} ((cost_j + \Delta cost_j) - \overline{cost'})^2 + \\ & \sum_{j \notin U_i, j \notin L_i} (cost_j - \overline{cost'})^2 \} \end{aligned} \tag{4.3}$$

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

Extending the optimization unit

We mentioned earlier that the scope of the optimization unit and the optimization rule set of XPORT can be extended. Here, we describe the impact of these extensions to the way XPORT quantifies the benefit of every optimization rule. Obviously, increasing the size of the optimization unit will simply increase the number of terms in Equations 4.1, 4.2 and 4.3, adapting them to include the extra nodes.

Defining new optimization rules is simple from the user perspective. The user simply defines which optimization rules will be combined along with the desired parameters. From the system perspective, defining a new optimization rule requires the definition of the equation that quantifies its impact. This equation can be derived by aggregating, using the system cost aggregation function, the equations for each of the optimization rules 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 optimization rules, 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.

Evaluating node cost changes

In order to quantify the benefit of an optimization rule we need to estimate its impact on its unit's nodes and dependents of the unit. If the system cost is calculated by additive or bottleneck functions, then the impact on the cost metric of the dependents equals to the impact on their aggregated cost, i.e., the dependence set cost (based on Equations 4.1 and 4.2). XPORT strives to incur the minimum communication overhead when estimating these cost changes. It allows communication among nodes of the same unit and avoids any metadata exchange with nodes lying outside the optimization unit. Instead, nodes maintain state for the dependents of their unit. We refer to this metadata as the *transformation state*.

XPORT exploits the semantics of the aggregation functions to identify both the minimum state required and to derive generic equations that quantify the cost change of the dependents of their unit. 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 defining the node cost, XPORT uses a brute-force approach for evaluating the impact of optimization rules. It calculates the new cost metric of *all* the dependents of the unit and uses Equations 4.1, 4.2 and 4.3 to quantify the result of an optimization rule. This approach may incur high communication overhead, as the metric definition does not include any aggregation function and XPORT's optimizations can not be applied. Thus, application designers should refrain from defining optimization metrics that do not adhere to the structured rules of our definition grammar.

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.

1. *Additive system cost functions.* 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.

2. *Bottleneck system cost functions.* In this case, 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. Since every dependent will have the same cost change, $\Delta cost_i$, then:

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

   Moreover, based on Equation 4.2, every node simply needs to maintain an estimation of the current system cost and its dependence set cost.

3. *Holistic system cost functions.* 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 peer in the system.

Given these equations, node n_i can estimate the benefit of an optimization rule using Equations 4.1, 4.2 and 4.3.

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 the local metric 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 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). The impact (if any) 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 metrics 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 metric and $h_i(j), \forall n_j \in D_i$, is referred to as the* minimum local metric *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 occurrence 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 an optimization rule on the system cost it needs to know the minimum cost of all system nodes min_{net}, except the ones affected by an optimization rule. This implies that every node needs to maintain the minimum cost of all nodes that do not belong to its dependents set.

We now describe the equations based on which n_i can calculate a change in its dependence set cost, when its cost change is $\Delta cost_i$.

1. *Additive system cost functions.* 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

   $$\gamma = \sum_{(\lambda_j, \alpha_j) \in T_i, \alpha_j \geq cost'_i} (cost'_i - \min\{cost_i, \alpha_j\}) \times \lambda_j$$

   In the first case, all the links in n_i's subtree have bandwidth capacity higher that n_i's path to the root. Hence n_i and its dependents share the same bottleneck link and since it is replaced by an even worst link the cost of *all* of them is affected that same, i.e., $\Delta cost_i$. The second option refers to the case where some dependents had a different bottleneck value before we applied the optimization rule, but are expected to share the same bottleneck link with n_i after the optimization rule is applied. The term γ identifies these nodes and aggregates their total cost change. These are the nodes that their links connecting them to n_i have a bandwidth capacity higher than then new bottleneck value of n_i. For all the other dependents there is not cost change and are not included in the equation.

In the second case, where $cost_i$ increases, the cost change on the dependent set cost is:

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

where

$$\omega = \sum_{(\lambda_j,\alpha_j)\in T_i, \alpha_j \geq cost'_i} \Delta cost_i \times \lambda_j + \sum_{(\lambda'_j,\alpha'_j)\in T_i, \alpha'_j < cost'_i \text{ and } \alpha'_j > cost_i} (\alpha'_j - cost_i) \times \lambda'_j$$

The first terms covers the case where all the links in n_i's subtree have higher bandwidth than the old *and* the new bottleneck link of n_i. Hence, *all* dependents will have the same cost change as n_i, since they share the same bottleneck link. In the case that only a subset of the dependents belong in the above category, we use the term ω two calculate the impact on two different sets of dependents. The first one includes the dependents that connect to n_i through links with higher bandwidth then the new bottleneck link of n_i. Thus, the total cost change of them is the same as for n_i. The second set includes the nodes that shared the same bottleneck link with n_i before the optimization rule. However, after this link was replace with the new one (with higher bandwidth capacity), their new bottleneck link is a link between n_i and the dependent and has a lower capacity than then new bottleneck link of n_i.

Given this value of $\Delta cost(D_i)$, n_i uses Equation 4.1 to estimate the benefit of an optimization rule. Thus, the state required is the size of the dependence set and the set T_i.

2. *Bottleneck system cost functions.* For the bottleneck functions, we are interested in the minimum change across 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, the impact on the dependence set cost is:

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

The first term refers to the case where n_i and all its dependents share the same bottleneck link. Since we assume that this value decreases, these nodes will experience the same decrease on their bottleneck value. The second term refers to the case where some dependent(s) has a different bottleneck link than n_i after the optimization rule. This happens when there exists a worse link between this dependent and n_i. However after the optimization rule, another link with lower bandwidth capacity is connecting n_i with the root of the tree, affecting the bandwidth bottleneck of the dependent as well. The third term refers to the case where the bottleneck value of all dependents is not effected. This occurs when there exists a link between any of the dependents and n_i with a worse bandwidth than then new bottleneck value of n_i.

Similarly, if $cost_i$ increases, then the impact on the dependent set cost is:

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

The first term refers to the case where n_i and its dependents share the same bottleneck link before *and* after we apply the optimization rule. This occurs when all the links between every dependent and n_i have higher capacity from the new and the old bottleneck link of n_i. The second term covers the case where there exists a link between some dependent and n_i that has a higher bandwidth capacity compared with the old bottleneck link of n_i, however, after this bottleneck value of n_i increases, this link becomes the new bottleneck link for some of the dependents. Finally, the third term refers to the case where bottleneck links of all the dependents lie within n_i's subtree. Since the bandwidth bottleneck of n_i increases, it could not affect the "worst" link of any of the dependents. Thus, none of the dependents' bottleneck value changes.

3. *Holistic system cost functions.* 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 of the node is:

$$\Delta cost_j = \begin{cases} \Delta cost_i & g_i \geq cost_i \\ \gamma' & g_i \leq cost_i \text{ and } h_i(j) \leq cost'_i \\ 0 & \text{otherwise} \end{cases}$$

where

$$\gamma' = cost'_i - \min\{cost_i, h_i(j)\}$$

In the first term, 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 has a different bottleneck value before the optimization rule, but shares the same bottleneck link with n_i after the optimization rule. Hence, its bottleneck link was lying between n_i and n_j before the optimization rule, while after we apply the rule it will be one of the links connecting n_i to the root. In all the other cases, there is no impact on n_j, i.e., there is a link between n_i and n_j that has lower bandwidth than the link between n_i and the root.

In the second case, where $cost_i$ increases, the impact on n_j's cost is:

$$\Delta cost_j = \begin{cases} \Delta cost_i & h_i(j) \geq cost'_i \\ h_i(j) - cost_i & h_i(j) \leq cost'_i \text{ and } h_i(j) \geq cost_i \\ 0 & \text{otherwise} \end{cases}$$

In the first case, the links between n_i and n_j have a higher bandwidth capacity than the old *and* the new bottleneck value of n_i, i.e., the bottleneck link lies on the path connecting n_i to the root and so the impact on n_j is the same as the impact on n_i. In the second case, n_j has a different bottleneck link than n_i, because there is a link between them with less capacity than the new increased bottleneck value of n_i. For all the other cases, there is no impact on n_j's cost.

4.3.4 Multi-metric cost functions

XPORT can create multi-metric overlay trees by defining either the node cost or the local metric 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. For example, let us assume an application defines the local metric of a node as the product of its incoming data rate and its latency to its parent. Each node will be customized to collect the required state and statistic for each of these metrics independently. Hence, nodes will measure latency and data rate on their link to their parent. Each node will combine these metrics, based on the application-specified function (the product in the specific example), to obtain the final local metric. In the same way, if the node cost is defined as the product of its latency to the root and its incoming rate, then each node will evaluate the latency to the root as a one-level aggregation metric and multiple it with the incoming rate, which could be defined as a local metric.

Moreover, for multi-metric node costs, the optimization procedure handle every metric independently. For each optimization rule, the node calculates the effect of the optimization rule on its dependents separately for each individual metric. It then combines these metrics to derive the final impact on its dependents' costs. In the previous examples, nodes will evaluate the impact on a node's latency from its parent (or on its latency from the root) and also quantify the impact of the optimization rules on the incoming data rate of each affected node. Based on these metrics, we evaluate the impact on the node's cost by combining them using the function provided by the application (in this example we multiple the new values). Since there are no restrictions on the combination functions, the total benefit of an optimization rule is calculated based on Equation 4.3, i.e., each node uses its estimation of the new costs for every affected node, calculates the new system cost and compares it with the current system cost value.

4.3.5 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 an optimization rule. In order for n_i to calculate this state, its peers need to periodically broadcast some metadata. This metadata includes the cost and local metric 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:

$$w_i \times p \tag{4.4}$$

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 metric 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 metric of n_i is to the cost of its dependents, the more likely it is to become their new bottleneck value (assuming no drastic changes on the local metrics of the nodes). Thus, we set the weight of n_i to $w_i = l_i - g_i$, where g_i represents the minimum (or maximum) bottleneck value of all the dependents of n_i and l_i its local metric. 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.

4.4 Run-time functionality

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

Node architecture

The high-level architecture of an XPORT node is shown in Figure 4.5. 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 optimization rules on the basis of application-specified performance criteria and constraints. The tree optimization rules 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 and metadata communication.

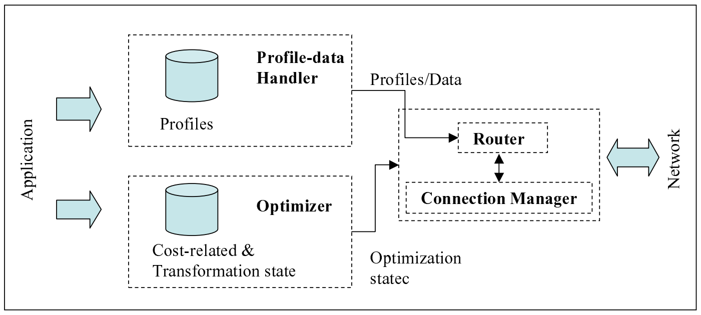


Figure 4.5: XPORT's high-level node architecture.

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 optimization process. This state allows nodes to quantify the cost benefit of their local optimization rules. Optimization state refers to the information nodes need to exchange with their neighbors during optimization and resides in the optimizer. Optimization and transformation state and their size estimations are given in Table 4.6.

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 4.6: Transformation state (T) and optimization state (O) for node n_i (V: set of nodes).

4.5 Network optimization protocol

As mentioned in Chapter 3, the local search heuristic used by XPORT follows the principles of a *hill-climbing* local search. Our optimization process identifies and applies the best network optimization across all possible optimizations of all units in the tree. This best network optimization rule will reorganize the structure of the dissemination tree. Our results revealed that this approach can yield significant improvements. Although this heuristic is quite prone to local optimum, our directed optimizations allowed XPORT to converge to network topologies with near-optimal performance.

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 and 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.

Simulated annealing

To show the effectiveness of our heuristic we compared it with another widely-used local search heuristic, *simulated annealing*. This approach has been shown to generate very efficient solutions for hard problems. The algorithm is illustrated in Figure 4.6. It starts from a random configuration C_0 and an initial temperature T_0. In the inner loop, a new configuration $newC$ is chosen randomly from the neighbors of the current configuration C. If the cost of $newC$ is smaller than that of C, the transition will happen. Otherwise, the transition will take place with probability of $e^{-\Delta Cost/T}$. (With the decrease of T this probability would be reduced.) Meanwhile, it also records the minimum cost configuration that has been visited. Whenever it exits the inner loop, the current temperature would be reduced. Based on previous related work [55], we select the parameters as follows: (1) $T_0 = 2 \times cost(C_0)$; (2) $frozen : T < 0.001$ and $minC$ unchanged for 10 iterations; (3) $equilibrium = 64 \times |N|$; (4) $reduceTemp = T \leftarrow 0.95T$; (5) $RandomRule(C)$: randomly choose one of the optimization rules of the configuration C.

```
begin
    $C \leftarrow C_0$; $T \leftarrow T_0$; $minC \leftarrow C$;
    while !$frozen$ do
        while !$equilibrium$ do
            $newC \leftarrow RandomRule(C)$;
            $\Delta Cost \leftarrow cost(newC) - cost(C)$
            if $\Delta Cost \leq 0$ then $C \leftarrow newC$;
            else $C \leftarrow newC$ with probability $e^{-\Delta Cost/T}$;
            if $cost(C) < cost(minC)$ then $minC \leftarrow C$;
        $T \leftarrow reduceTemp(T)$;
    return $minC$;
end
```

Figure 4.6: Simulated annealing algorithm.

Concurrent optimization rules

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

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

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

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

If two optimization rules are applied on independent units, then they cannot negatively interfere with each other. Otherwise, one of the optimization rules 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 optimization rules 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 optimization rule. 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 optimization rules 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:

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

4.6 Performance evaluation

We implemented an initial XPORT prototype [43] 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 into 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 [51] possesses a similar structure, though it is unable to perform XPORT's wide variety of optimizations.

We studied XPORT's performance through real-world RSS data and results from simulation, prototype-based LAN emulation and deployment on the PlanetLab testbed. In our PlanetLab experiments, we used dissemination trees with up to 200 randomly chosen PlanetLab sites. For our prototype-based LAN emulation, we used up to 200 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 and maximum network latency. Our experiments demonstrate XPORT's flexibility and effectiveness, as it manages to improve each of these metrics significantly through its local optimization rules.

In our experiments we study three different variations of our optimization framework:

- *XPORT-local* uses only local optimization rules,
- *XPORT-directed* uses only directed optimization rules and
- *XPORT-hybrid* uses both types of rules.

4.6.1 Convergence

In our first set of experiments, we study the convergence properties of XPORT. Ideally, we would like to have a system that converges to the optimal configuration using a small number of optimization steps. To measure the efficiency of our approach, we identified the optimal topology for certain metrics and compared its performance with the best configuration achieved by XPORT. The results

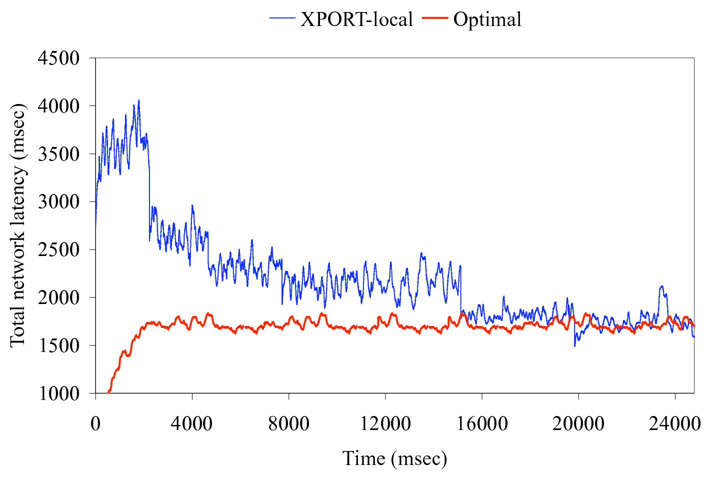


Figure 4.7: Convergence to the optimal for the total network latency.

show that XPORT can achieve near-optimal performance for most of our metrics and within small number optimization steps.

To demonstrate the efficiency of our optimization rules and our hill-climbing optimization local search (i.e., at each step, we apply the best optimization rule), we compared XPORT with the performance of the simulated-annealing local search, as described in Figure 4.6. We compared XPORT to two variations of the simulated annealing approach:

- *SA-local*, where the function $RandomRule(C)$ picks a random rule from the set of local optimizations.
- *SA-hybrid*, where the function $RandomRule(C)$ picks a random rule from the set of local *and* directed optimizations.

Average network latency

Figure 4.7 shows the sum of the network path latencies of 40 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 this topology, all the nodes are connected directly to the root of the tree. XPORT starts with a random tree and continuously applies local optimization rules. The figure shows that XPORT starts with lower performance than the star topology and after a small number of optimization rules our tree converges to the optimal tree.

We also run experiments demonstrating the scalability of our system and its ability to converge to optimal configurations for large networks. Figure 4.8 shows the average network latency for different

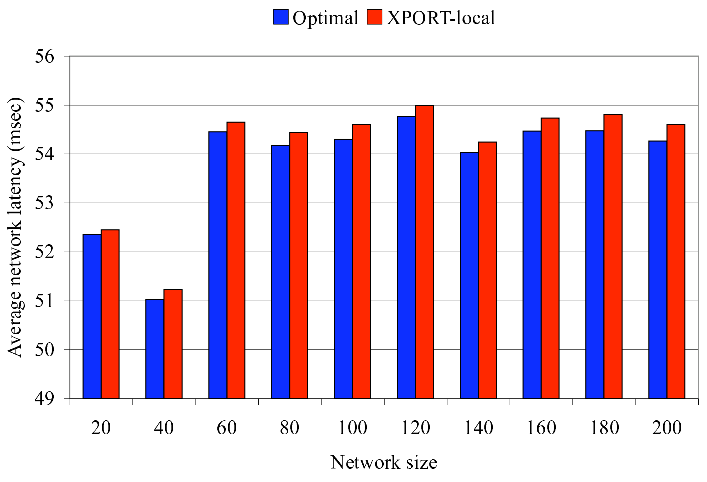


Figure 4.8: Convergence to the optimal for the average network latency.

	20	40	60	80	100	120	140	160	180	200
XPORT-hybrid	16.6	39.6	61.1	83.6	103.8	124.9	151.3	174.7	188.7	210.7

Table 4.7: Number of network transformations for optimizing the average network latency.

network sizes. The results were obtained from our LAN emulation and show the average performance achieved over fifty (50) different runs, where at each run XPORT is initialized to a different random tree. The optimal was obtained by the Dijkstra algorithm. The results show that XPORT is able to converge very close to the optimal configuration for all network sizes. Note that for this metric, local optimizations are sufficient to converge to the near-optimal configurations. Table 4.7 shows the number of optimization rules applied by XPORT before it converged to the results of Figure 4.8.

We also tried to optimize XPORT for the network latency, but under certain constraints; we applied restrictions on the number of children each node could have in the XPORT tree (referred as fanout). Table 4.8 shows the results for this metric for a network of 20 nodes. When no constraints are imposed, XPORT can converge to the optimal solution. As the constraints become tighter, fewer optimization rules can be applied and smaller improvement is achieved.

Redundant data

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 using only local optimizations and also to show the effectiveness

Metric	f	t	**Performance**
Total path latency	2	5	3229ms
	6	8	3036ms
	10	12	2670ms
	∞	14	1691ms (0.02% off the optimal)
Redundant incoming data	2	4	1088KB
	6	7	461KB
	10	8	304KB
	∞	11	0KB (optimal)

Table 4.8: Performance for different constrained metrics (t: number of optimizations, f: fanout constraint).

of our optimization rules.

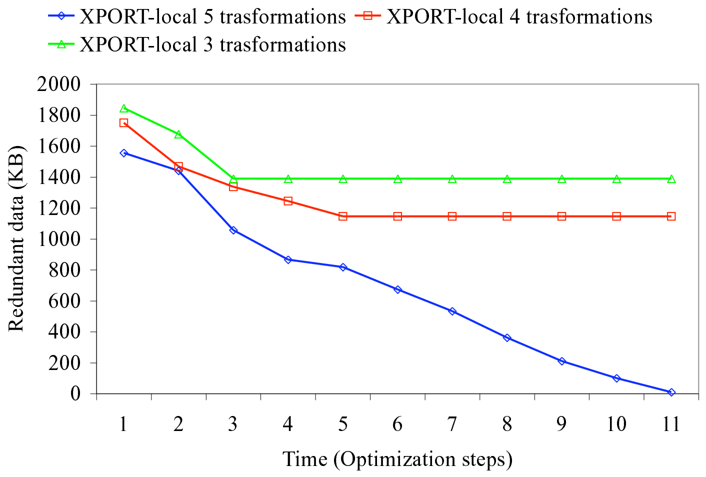


Figure 4.9: Convergence to the optimal for the average superfluous incoming data.

For this metric we used our RSS feed application on 40 PlanetLab sites. Figure 4.9 shows three different cases for this metric, one when all five composite optimization rules are applied and two where the optimizer used only four and three composite optimization rules. We first remove the promote subtree optimization rule and then the parent-child swap optimization rule from our optimization. The results reveal that in the first case XPORT converged to the optimal solution (where the total redundant data is zero for every node) after applying 11 optimization rules, while for the other two cases the system cost improves but could not converge to the optimal case. Promote subtree is the most beneficial optimization rule because it allows our tree to converge to the star topology. XPORT managed to improve its performance, even in the absence of this optimization rule. Moreover, the larger the set of optimization rules, the better performance XPORT achieves.

We also implemented a constrained version of this metric, by defining constraints on the fanout of the nodes. Table 4.8 shows the results for the constrained metric. Similar to previous constrained metrics, when no constraints are imposed, XPORT can converge to the optimal solution. However, the tighter the constraints we express, the fewer optimization rules can be applied and smaller improvement is achieved.

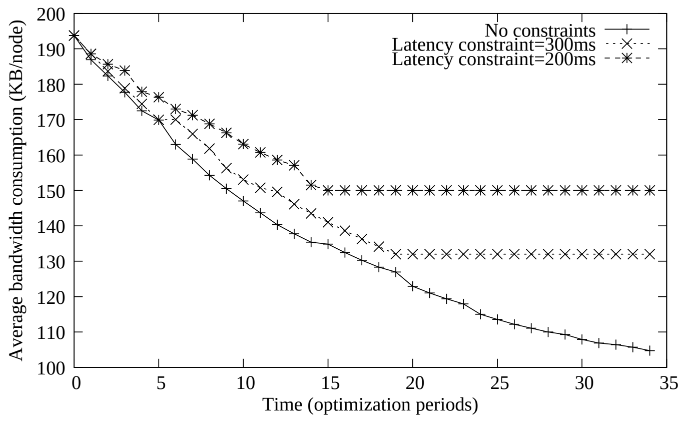


Figure 4.10: Convergence to the optimal for the average bandwidth consumption under various latency constraints.

Bandwidth consumption

We studied also metrics where a node's cost is aggregated over its children's local metrics. An example metric is 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. This metric includes the CPU time for processing and matching the incoming messages and is implemented as an 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 4.10 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, using again only local transformations. 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.

Bandwidth bottleneck

We also run experiments trying to maximize the bandwidth bottleneck. This metric 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.

From our experiments, we discovered that this metric was very sensitive to local optimum and we had to adjust our optimization approach in order to provide network configurations with performance close to the global optimum. Figure 4.11 compares XPORT with the optimal topology for different network sizes. XPORT was initialized to a random initial tree and its optimization rules were continuously applied until it converged. We repeated the experiments fifty times starting each run from a different initial tree and we show the average performance it converged to over all 50 runs. The figure shows the difference between the performance of the optimal topology and XPORT. Localized transformations were only able to improve the bandwidth bottleneck by up to 10-15%, but eventually XPORT converges to a local optimum. Directed optimizations (combined or not with the local ones) can lead to the global optimum for small network sizes and they can achieve an improvement around 60% from the initial configuration. However, as the network size increases to more than 60 nodes directed rules still converge to local optimum.

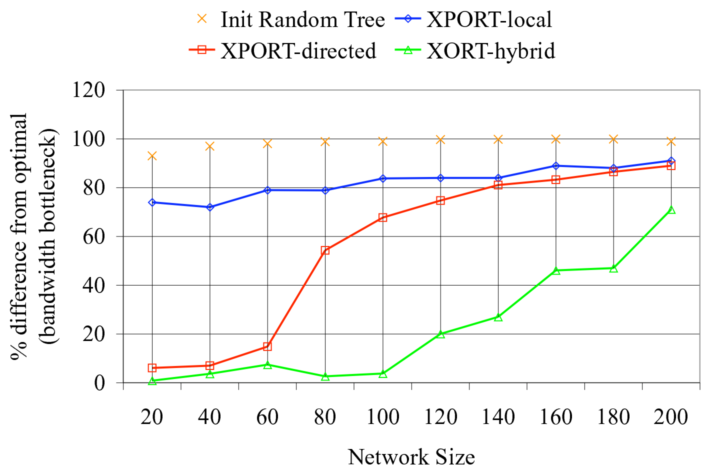


Figure 4.11: Convergence to the optimal for the bandwidth bottleneck metric and a random initial tree.

Figure 4.12 shows the number of optimizations applied in order to achieve the convergence results of Figure 4.11. The graph reveals that as we increase the network size XPORT is unable to discover beneficial directed optimizations and escape the local optimum. Thus, in a larger network, XPORT converges much faster to a local optimum.

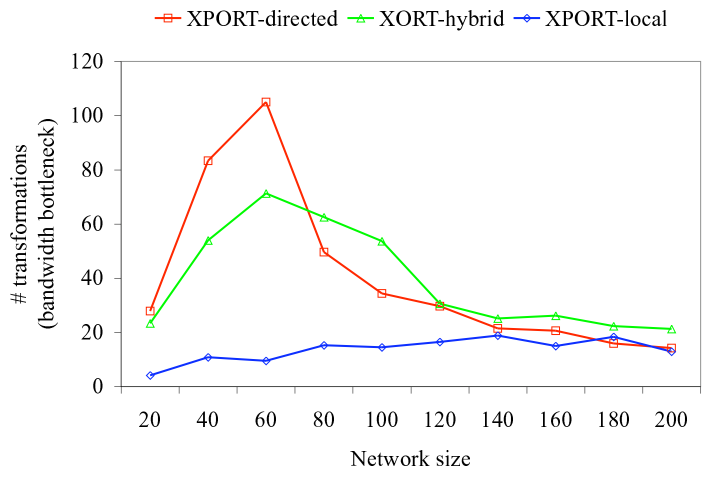


Figure 4.12: Number of network transformations for optimizing the bandwidth bottleneck metric and a random initial tree.

One interesting result we obtained for the bandwidth bottleneck metric is that the optimization rule that yields most of the improvement is the directed swap. Figure 4.13 shows how XPORT-hybrid compares to the optimal when we remove the swap optimization from the available rules and when directed swap is allowed. Based on the results, directed swap can increase XPORT's performance by up to 25%.

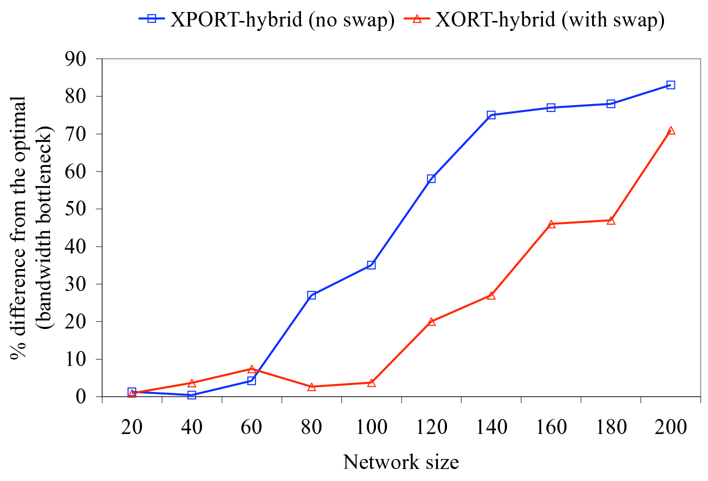


Figure 4.13: Impact of directed swap operations for optimizing the bandwidth bottleneck.

In order to avoid convergence to local optimum, we applied the following heuristic. Instead of starting from a random initial tree, we configured XPORT to create a best-effort tree for its initial configuration. In this case, when a node joins the tree it picks as a parent the available node that maximizes its bandwidth consumption. We run 50 different experiments, changing every time the order in which nodes join the system. Figure 4.14 shows the difference between the optimal and the average performance XPORT can achieve for the two different initial trees. The results show that starting with the best-effort tree, XPORT can converge to a configuration within 0-7% from the optimal. Moreover, as we increase the network size the performance does not degrade (as in the case of an random initial tree) but it constantly provides configurations that do not perform more than 7% compare to the the optimal. Hence, our simple heuristic allowed XPORT to achieve near-optimal performance.

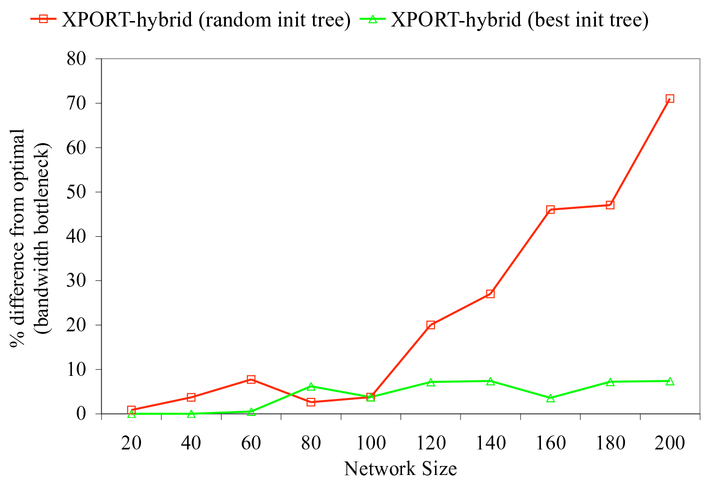


Figure 4.14: Convergence to the optimal for the bandwidth bottleneck and different initial trees.

We were able to understand better the above result after we visualized the structure of the optimal tree and the topologies provided by XPORT. In the optimal tree a small number of nodes with high bandwidth connections to the root of the tree have a very high number of children. At the same time the vast majority of the nodes have very few, most often zero, children. We can think of the optimal network as a network of clusters connected with high-bandwidth links, while the topology of each cluster is a near-start topology. On the other side, when XPORT is initialized to a random tree, it creates trees with small fanout and high height. As the network size increases, nodes joining the system last have a large set of nodes they can be as their parent, resulting in long, unbalanced paths. A best-effort tree corrects this topology and initializes XPORT to a network that resembles that of the optimal topology. Nodes tend to cluster around the existing nodes with the highest available bandwidth. During optimization, directed swap operations, identify which low bandwidth links should be replaced with better ones and lead XPORT close to the optimal configuration.

Figure 4.15 shows the optimization rules applied for the results shown Figure 4.14. In most of the cases, starting from the best-effort tree required more optimizations in order for XPORT to converge. This is explained from the fact that XPORT escapes local optimum and thus, continuous to consider alternative optimization rules. XPORT applies around 40% more optimizations but also converges to configurations that perform up to 70% better than in the case of a random initial tree.

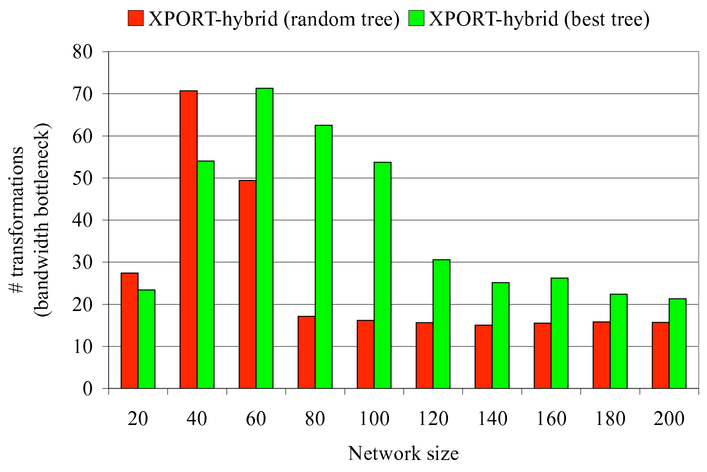


Figure 4.15: Number of network transformations for optimizing the bandwidth bottleneck and for different initial trees.

As the next step, we studied how important directed optimizations are when the best-effort tree is used as XPORT's initial tree. Figure 4.16 compares the average performance achieved by our three XPORT variants with the optimal cost, over 50 different runs for different network sizes. One important observation is that, although XPORT is initialized to the best-effort tree, this configuration performs around 50-60% worse than the optimal. Local optimizations improve the performance by up to 30%, however they lead to networks that are 20-25% worse that the best possible. Using directed optimizations allows for an extra 20% improvement on the performance and XPORT converges to the a global optimal (for small networks) or to a configuration that is only 10% worse than the optimal. For this metric, localized transformations do not provide a significant improvement and XPORT relies solely on the directed ones, whenever available. Hence, XPORT-directed and XPORT-hybrid have similar performance.

Figure 4.17 shows the number of optimization rules required to achieve the results of Figure 4.16. XPORT-directed and XPORT-hybrid apply almost the same number of transformations. They require more optimizations steps than XPORT-local, however they converge to a considerably better configuration. Moreover, as network size increases less optimizations are required. The important

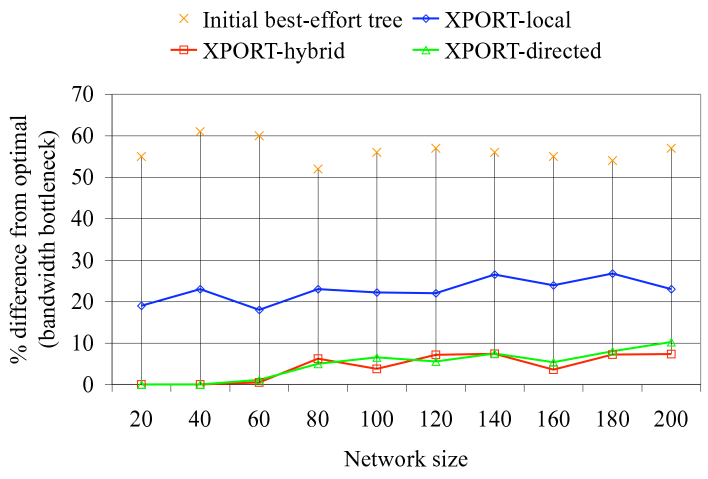


Figure 4.16: Convergence to the optimal for the bandwidth bottleneck and a best-effort initial tree.

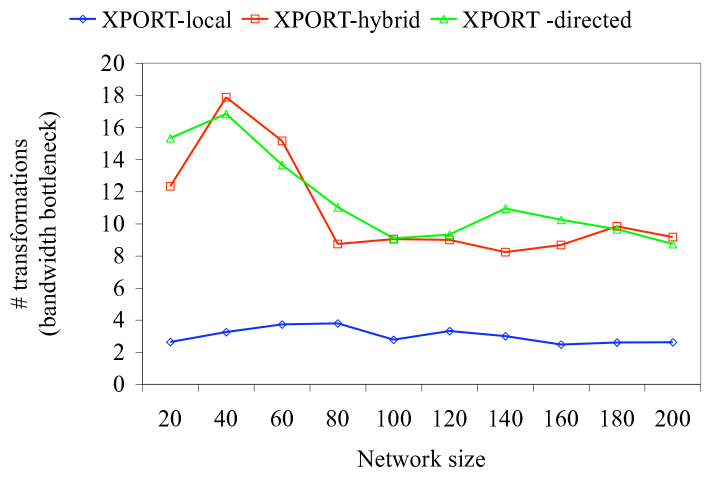


Figure 4.17: Number of network transformations for optimizing the bandwidth bottleneck metric and a best-effort initial tree.

observation here is that across all network sizes XPORT does not require more than 20 optimization rules in order to discover the optimal configuration.

Figure 4.18 compares XPORT's local search with the simulated annealing search. When only local optimization rules are allowed both approaches cannot converge to the optimal configuration. The simulated annealing converges to configuration that is around 10% off the optimal. XPORT's performance is worse, being 15-23% from the optimal. However, when directed optimizations are allowed, both approaches converge to the optimal solution. Table 4.9 shows the number of optimization steps required by these heuristics. While XPORT needs less than 80 optimization rules to discover the optimal, the simulated annealing approach requires thousands of random reconfigurations.

We also studied the bandwidth bottleneck metric under fanout constraints that limit the children

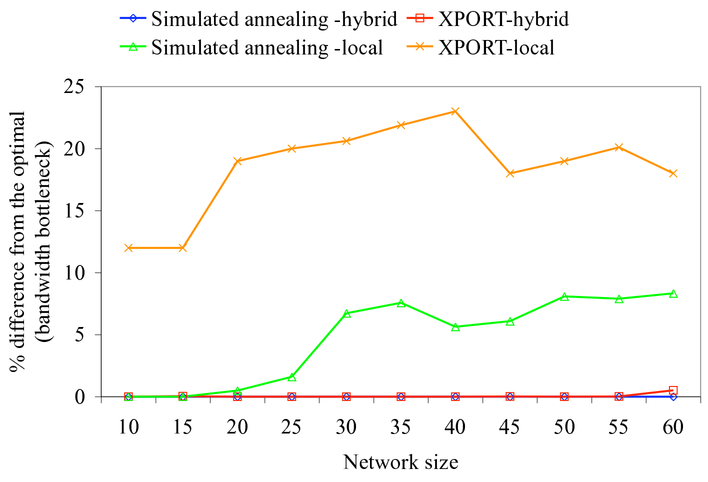


Figure 4.18: Convergence to the optimal for XPORT and simulated annealing and the bandwidth bottleneck metric.

	10	20	30	40	50	60
XPORT-local	1.56	3.5	3.2	3.84	3.5	3.92
Simulated annealing-local	9,234	11,4545	50,232	62,7676	93,535	118,334
XPORT-hybrid	3.16	7.24	10.42	14.12	13.62	71.3
Simulated annealing-hybrid	18,721	45,1256	99,591	132,567	198,456	234,932

Table 4.9: Number of network transformations for XPORT and simulated annealing and the bandwidth bottleneck metric.

of every broker to a constant number. These results are from our prototype-based LAN emulation on 40 nodes. Figure 4.19 shows that XPORT identifies critical portions of the tree and after every optimization rule 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 optimization rules 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.

Maximum network latency

Finally, we studied the convergence property for the maximum network latency metric. Similar to the bandwidth bottleneck, this metric was also prone to local optimum. Figure 4.20 compared the three XPORT variants with the optimal configuration. In this metric, a combination of local and directed optimization appears to be very important in order to achieve good performance. Only when directed optimizations are used, XPORT is not able to perform well and is stuck in a local optimum. Localized optimizations are more effective. However, only when we combine directed and

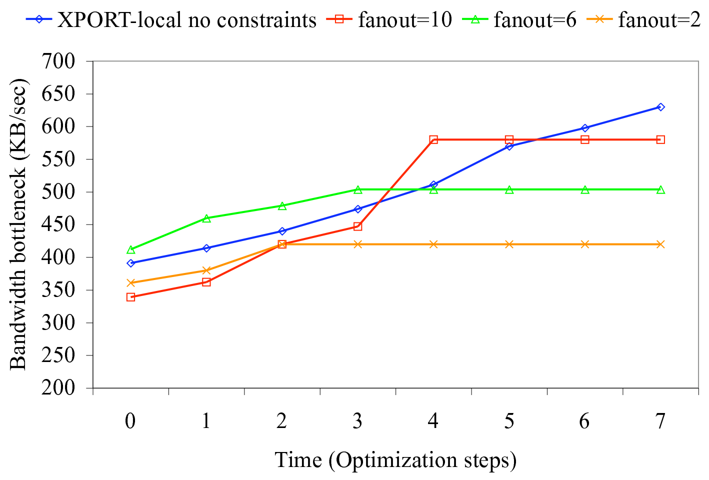


Figure 4.19: Bandwidth bottleneck under fanout constraints.

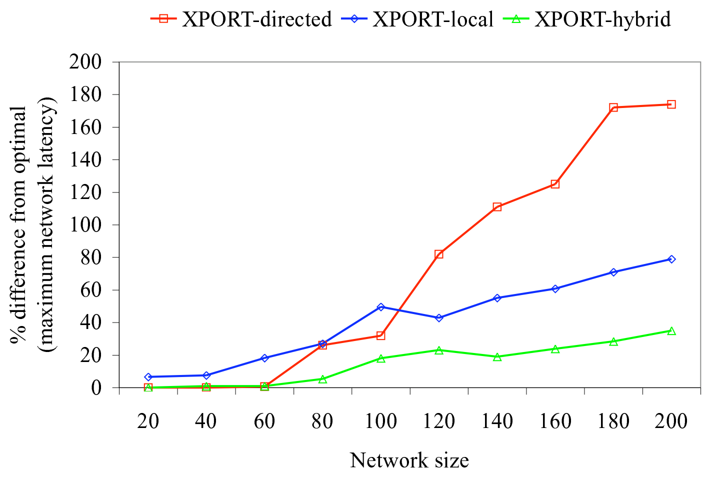


Figure 4.20: Convergence to the optimal for the maximum network latency.

local optimization rules we maximize XPORT's performance. It converges to the optimal network for small networks and within 20% from the optimal for networks with size between 100-200 nodes.

Figure 4.21 shows the number of optimization rules applied in average. Considering only localized or directed optimization rules does not allow XPORT to discover possible optimizations and convergence to local optimum. By using both types of rules XPORT converges close to the optimal with a small overhead on the number of required reconfigurations.

Figure 4.22 compares XPORT with the simulated annealing approach. XPORT performs closer to the optimal than simulated annealing, even when using only the localized optimizations. Moreover, when directed rules are also used it converges to the optimal. Similar performance is observed for XPORT as we increase the number of available nodes, while simulated annealing decreases the

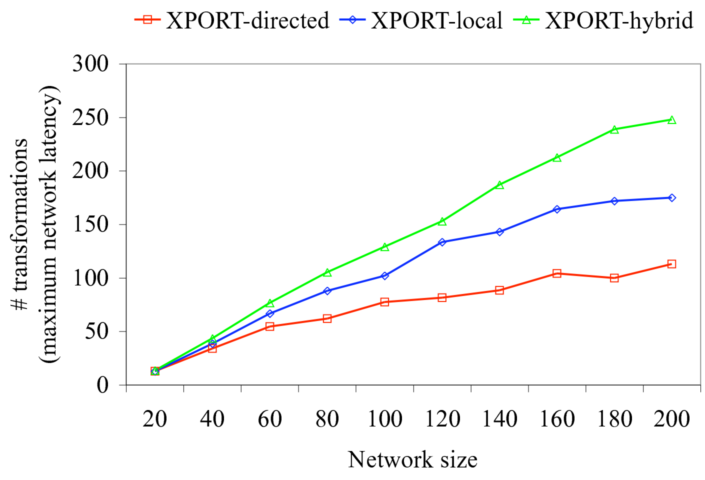


Figure 4.21: Number of network transformations for optimizing the maximum network latency.

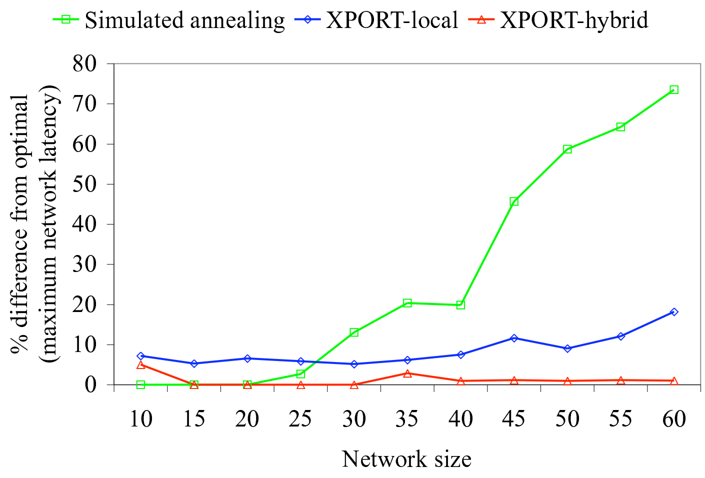


Figure 4.22: Convergence to the optimal for XPORT and simulated annealing for the maximum network latency metric.

performance as the network size increases.

4.6.2 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 experiments optimizing 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

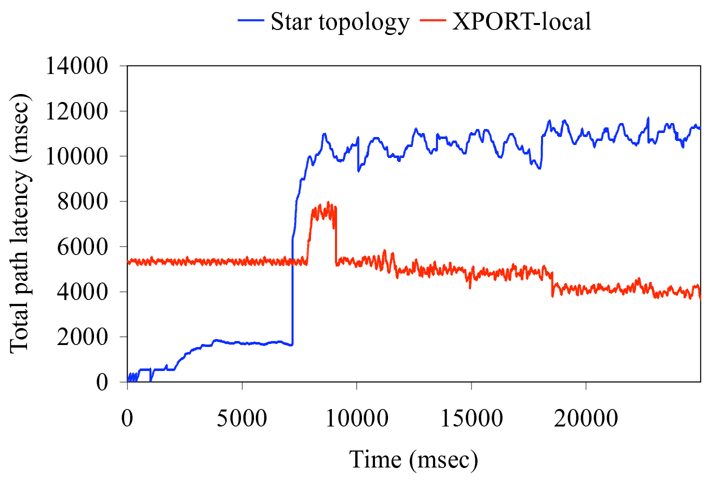


Figure 4.23: System adaptivity for the dissemination and processing latency.

across multiple nodes.

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 4.23. 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 optimization rules, 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.

4.6.3 Optimization Scope

In this section we study the impact of the optimization unit's size on the performance of XPORT. The results in this section are based on a simulation. We simulated three variants of XPORT-local, each using an optimization unit of different size. The first one uses a unit with size up to three levels (node, children, grandchildren). In this case we use the optimization rules described in Section 4.3.1. The second variant uses a unit of up to four levels. To exploit the scope of this unit we added four more local optimization rules:

- promote subtree 3 levels up: it promotes a node's subtree under its grandparent,

- demote child 3 levels down: it demotes a node (and its subtree) under one of its siblings' children,
- swap grandparent-child: it swaps a grandparent with one of its grandchildren and
- demote subtree 3 levels down: it demotes a node's subtree (but not the node itself) under one of its siblings' grandchildren.

Finally, we simulated a variant with an optimization unit of up to five levels. In this case, we extended the set of local optimizations with the following rules:

- promote subtree 4 levels up: it promotes a node's subtree under its grand-grandparent,
- promote node 4 levels up: it promotes a node under its grand-grandparent,
- demote child 4 levels down: it demotes a node (and its subtree) under one of its siblings' grandchildren,
- swap parent-grand-grandchild: it swaps a parent with one of its grand-grandchildren and
- demote subtree 4 levels down: it demotes a node's subtree (but not the node itself) under one of its siblings' grand-grandchildren.

We studied XPORT's convergence when optimizations with different scope are used. We only utilize local optimizations and the metric we used is the average network latency. Note that this metric can converge to the optimal by utilizing only local optimization rules. In the next paragraphs we refer to the XPORT variants we compare based on the scope of the rule it utilizes. Hence, "Level=4" refers to the case where only rules with scope of four levels are used, while "Level=3&4" implies that the optimization uses the rules defined over three levels *and* the rules defined over four levels.

Figure 4.24 shows the impact of the new optimization rules we defined on XPORT's performance. We note that in all the cases XPORT performs almost the same to the optimal (only 0.6% worse than the optimal in the worst case). The first observation is that as the network size increases XPORT performance slightly degrades. However its distance from the optimal is less that 1%. Extending the scope of our unit and defining more optimization rules seems to have a small impact as the performance improves only by 0.1-0.4%. Extending the scope of the unit above 4 levels does not allow for more improvement.

Figure 4.25 shows the number of optimizations required for convergence. Extending the optimization unit seems to have small benefit on the number of applied rules. As the network size increases,

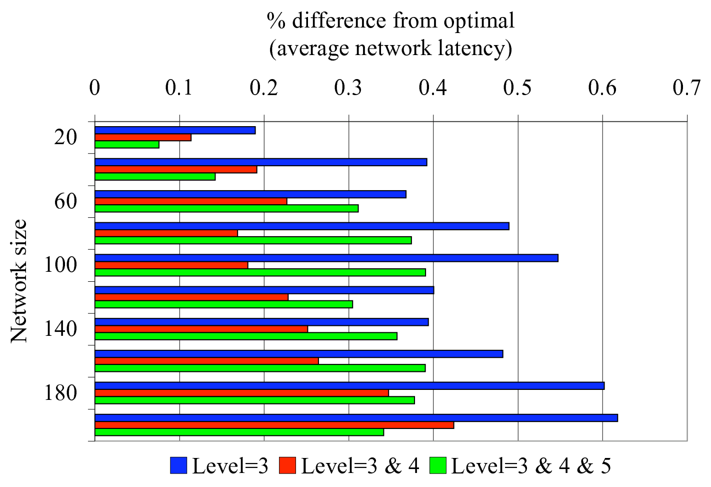


Figure 4.24: Convergence to the optimal for the average network latency and optimization units of different size.

more optimizations with larger scope can be applied reducing the required reconfigurations by 10 moves in average.

Figures 4.26 and 4.27 reveal the importance of fine-grained optimization rules. When rules defined over a smaller scope are removed from our system, XPORT cannot converge to the optimal. In the contrary, the more coarse-grained the network transformations, the worse XPORT performs. 5-level rules yield a performance that is 20-85% off from the optimal, depending on the network size. For smaller networks the performance is worse, as the trees do not have enough levels to apply the 5-level optimization rules. When 4-level rules are used the performance is better, but still is around 10% that the optimal. Only when these rules are combined with the 3-level rules can yield comparable to optimal performance, as shown in Figure 4.27.

4.6.4 Network traffic

Figure 4.29 shows the average maintenance traffic and the optimization traffic for optimizing different metrics in 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 optimization rules and it's size depends on the optimization metric. 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

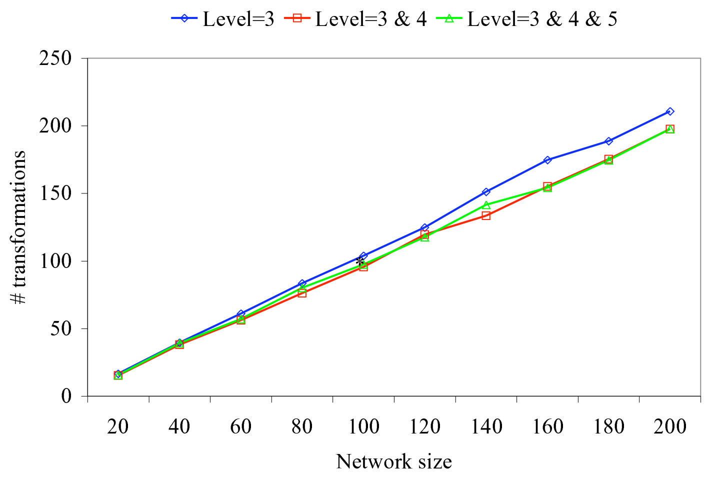


Figure 4.25: Number of network transformations for optimizing the average network latency and using optimization units of different size.

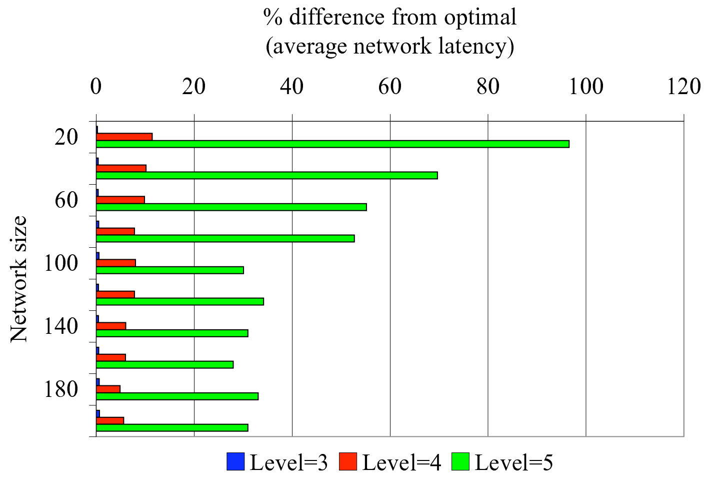


Figure 4.26: Impact of optimizations with small scope for optimizing the average network latency.

one level of aggregation, as the 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 optimization rules 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 exchange data. As a result,

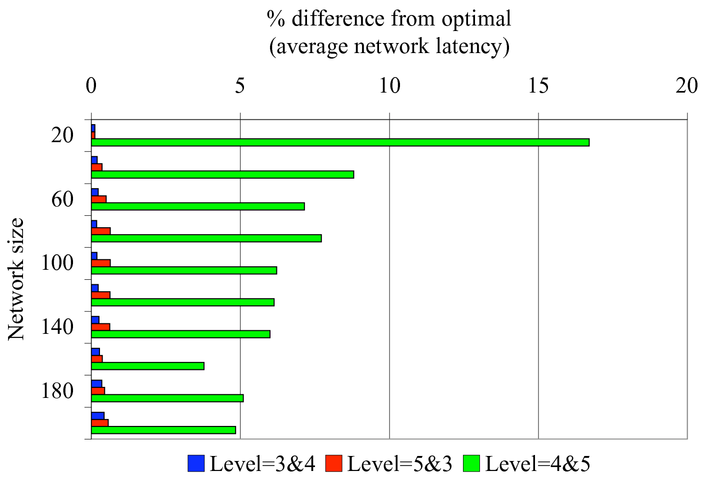


Figure 4.27: Impact of combining optimizations with different scope for optimizing the average network latency.

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 during optimization.

Statistics Approximation

We now study the tradeoff between traffic improvement and system performance when approximating statistics. For this purpose, we ran experiments 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 metrics are approximated. We expect that the more frequently nodes broadcast their cost values, the better cost estimations they can compute, leading to more effective optimization rules and thus faster convergence. Of course, if nodes broadcast their costs more frequently, the maintenance traffic will be higher.

The results in Figure 4.30 and 4.30 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 optimization rules. 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 4.4, 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

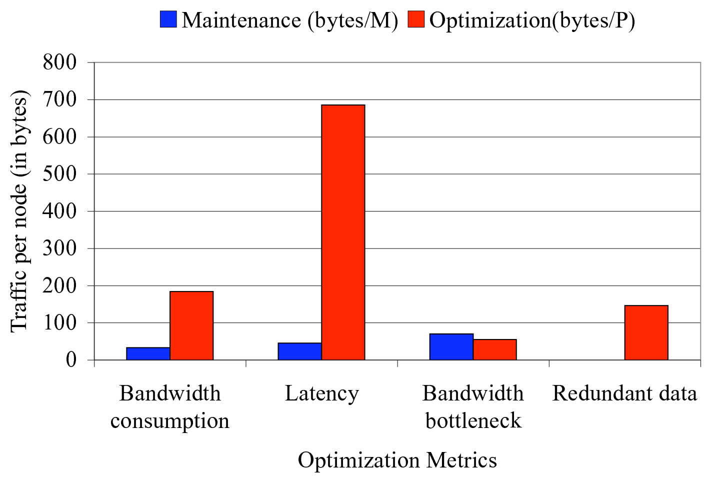


Figure 4.28: Maintenance and optimization traffic (S = Maintenance period, P= optimization period).

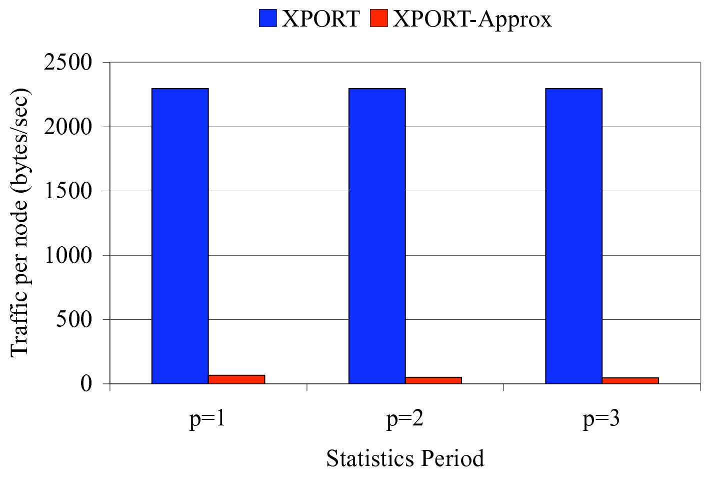


Figure 4.29: Improvement on network traffic due to statistics approximation.

values ($p = 1$), approximation reduces the maintenance traffic by 97%. At the same time, it takes 20 optimization rules 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.

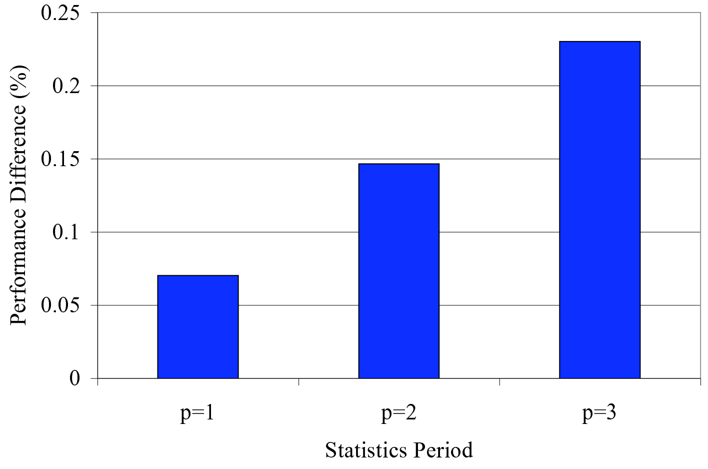


Figure 4.30: Impact of statistics approximation on performance when optimizing the path latency variance. The percentage values indicate relative differences over the non-approximated case

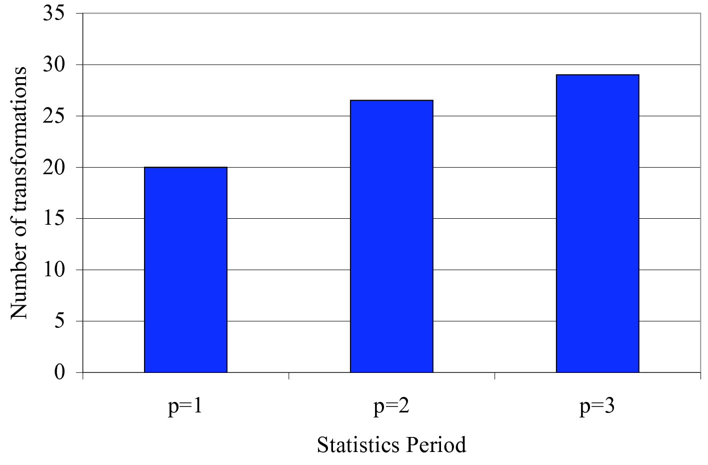


Figure 4.31: Impact of statistics approximation on convergence time when optimizing the path latency variance. The percentage values indicate relative differences over the non-approximated case

Chapter 5

Supporting Distributed Stream Processing Applications

While dissemination-based applications allow for simple profiles types (such as basic predicates) many large-scale applications require 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”. Therefore, to support applications with advanced processing requirements, we provide a “native” profile language that consists of stream-oriented operators [7, 42] which can perform filtering and merging, as well as complex time-window based aggregation and correlation over data streams. As such, users can express continuous stream processing queries as their profiles using either a graphical data-flow notation or a textual notation, both of which are already supported by Aurora and Borealis [6]. To efficiently support and optimize these stream-based queries, we designed and implemented *XFlow*, an ISM system that can be easily customized to support application-specific processing logic, performance expectations, and constraints.

XFlow builds on top of our generic dissemination infrastructure and extends it with stream processing functionality. It provides an extensible system for distributing and optimizing stream processing queries. It creates, maintains and optimizes an overlay network, given dynamic stream sources, clients with stream-oriented queries and application-specific performance expectations. The network consists of multiple, potentially overlapping dissemination trees, created dynamically depending on the degree of stream sharing. XFlow employs a unique combination of operator migration, replication and partitioning, and progressively refines the processing of the queries, the structure of the overlay trees, as well as the statistics collection process, to meet the desired objectives.

XFlow maintains the key feature of our infrastructure: it relies on *localized* state and interactions to reduce the *global* system cost. It utilizes and extends our basic aggregation-based metric definition model that allows us to rely only on localized, aggregated, network and workload statistics. Nodes distribute their queries on their neighborhood as well as on specific promising network regions which they discover through dissemination of these statistics. One of our key results is that even simple aggregations of statistics are sufficient to achieve efficient operation with low overhead, as well as allow XFlow to avoid local optimum and converge to near-optimal configurations. Moreover, we employ probabilistic statistics dissemination techniques that manage to keep the network traffic within constant bounds independent of the number of queries and nodes with low performance degradation. This characteristic allows for high scalability and efficiency.

XFlow relies on our basic publish-subscribe model for its underlying communication. As mentioned in Section 2.2, this model effectively decouples sources and destinations over geography and time: sources publish their data without knowing where and when he consumers will access them and consumers subscribe their processing queries without knowledge of specific sources. It is the responsibility of the system to collect the data and process the queries and distribute the results to the clients, while meeting application-specific performance expectations. This flexible, loosely-coupled architecture allows for scalable dissemination of high stream volumes to large number of consumers and robustness in the presence of high query subscription/unsubscription rate.

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

From an operational perspective, XFlow creates, maintains and optimizes an overlay mesh that consists of multiple, potentially overlapping publish-subscribe dissemination trees, in principle one for each source. XFlow extends our basic generic cost model to reflect this new network topology: it can express performance metric for multi-tree overlays. XFlow uses a set of optimization rules that modify the placement and execution of the query operators to reduce the system cost and meet constraints. These rules apply operator distribution operations, which include migration, replication and partition.

The main contributions of XFlow are the following:

- It uses a novel architecture that tightly integrates stream processing and our basic publish-subscribe stream routing model. The system consists of multiple, potentially overlapping overlay trees, for data collection, processing, and result dissemination.

- It provides a *generic* cost model that can express a range of cost metrics and constraints for evaluating the overhead and efficiency of *user queries* as well as various *resource utilization* metrics. The model relies on statistics aggregation in order to reduce the required state and network traffic.

- We describe a generic distributed *query optimization* framework that uses local, aggregated state and dynamically modifies the structure of the overlay as well the placement and processing of the operators through a set of migration, partition and replication operations. To the best of our knowledge, XFlow is the first system to allow this combination of query optimizations.

- We introduce a metric-independent statistics selection and dissemination mechanism that utilizes the semantics of the cost functions to identify available network resources. Based on this approach, we can carefully target our optimization towards low-cost, promising network regions and avoid local optimum.

The rest of the chapter is structured as follows. We describe the system model of XFlow in Section 5.1. We introduce our cost model in Section 5.3 and describe our generic optimization in Section 5.4, while we present our experimental results in Section 5.5.

5.1 System model

XFlow consists of an overlay network of cooperating *brokers* (or nodes), providing stream routing and stream-based query processing services (e.g., [7, 42]). *External data sources* reside outside system boundaries and publish data streams according to a well-defined global schema. Clients are also external and are eventual consumers of query results: they subscribe their interests expressed in terms of stream-oriented continuous queries on the global schema. Each external source and client has a proxy running on a broker, acting on behalf of its corresponding entity. In the rest of the discussion, we use node and broker, interchangeably.

5.1.1 Publish-Subscribe model

XFlow uses a uniform publish-subscribe mechanism for disseminating *all* data flows in the system. One implication is that each query operator publishes its output stream(s), while subscribing to its input stream(s). As operators also publish data, we refer to them as *internal sources*. Both external and internal sources are assigned system-wide unique identifiers and have well-defined schemas. The global schema is the union of external and internal schemas.

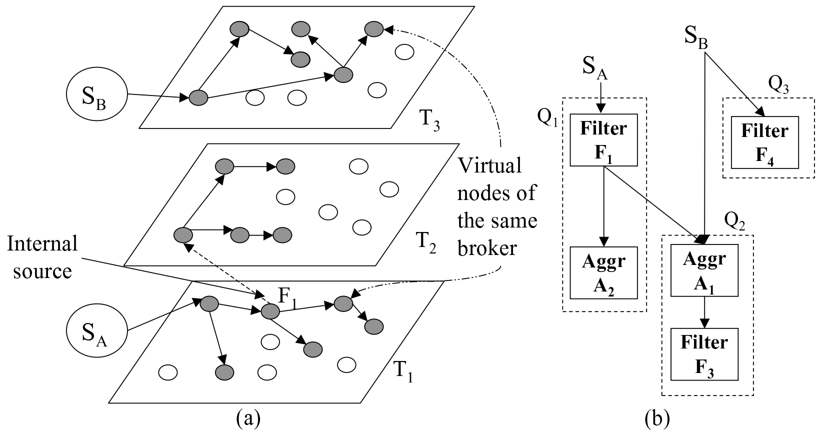


Figure 5.1: XFlow system model.

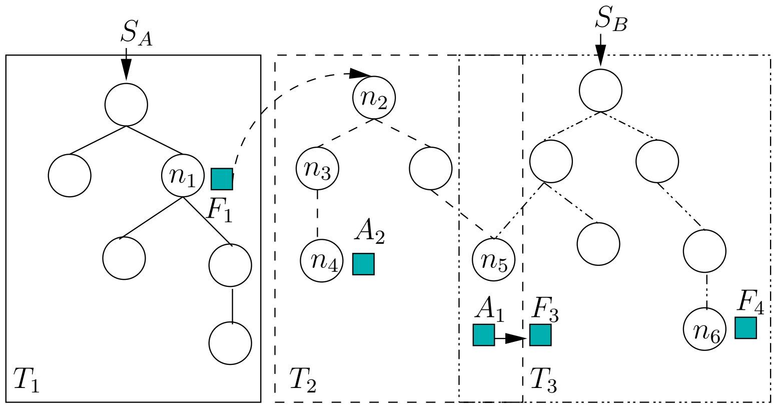
XFlow creates multiple overlay publish-subscribe dissemination trees, potentially one for each internal or external source. Each source publishes its stream to an overlay tree that distributes its data to the subscribed consumers (i.e., operators or external clients). Hence, nodes hosting clients or operators that are interested in a stream must join its corresponding dissemination tree. Figure 5.1(a) shows a network of three trees: T_1 and T_3 publish the external sources' streams S_A and S_B respectively, while T_2 publishes the output of the internal source F_1.

5.1.2 Source registration

When a source (internal or external) is first registered, it forwards its stream's schema to the registration service (referred to as *bootstrap node*). For an external source, the bootstrap is contacted through its proxy. The bootstrap picks a broker as the *root broker* for that stream based on its topological distance to the source, the available bandwidth of the broker and the expected data volume. This broker will be the root of the tree that disseminates the source's stream. Upon receipt of the first subscription for the data stream, the root broker will activate the data source, which in turn starts publishing its data streams. The functionality of the bootstrap can be easily distributed across multiple nodes to improve the availability and scalability.

5.1.3 Query registration

Clients also pick one XFlow node to host their proxy. To subscribe, the client's host contacts the bootstrap and requests a list of the streams published by both external and internal sources. Users can inspect the streams' schemas and define their queries on any collection of external or internal sources. This allows for intermediate results to be shared by multiple queries. For example, in



(c)

Figure 5.2: Distribution of queries in Figure 5.1

Figure 5.1(b) query Q_2 uses the output of operator F_1 of query Q_1. Following the model of the Aurora/Borealis [6, 16] and Yahoo!Pipes [5], XFlow allows users to browse the deployed query network in order to identify shared interests with existing queries and reuse already running processing operators. However, XFlow can be easily extended to incorporate techniques for automatic detection of sharable computations [40].

The bootstrap also informs the client's host about the tree(s) publishing its query's input(s). The host joins each of these trees, by selecting as parent one of the existing nodes in the tree. The choice of the parent depends on the application's performance expectations and constraints, e.g., for latency-sensitive applications it will be the closest node while for applications with fanout restrictions we would pick the node with the smallest number of children. Once the host joins the tree, it requests the query's input streams published by that tree. We refer to this request as the *profile* of that node and is extracted from the selection predicates of the query. This profile is forwarded upstream to the root, creating a reverse routing path to the node. Using the routing tree created, a node can route streams published through this tree only to descendants interested in receiving them.

5.1.4 Stream processing model

User profiles are expressed as directed, acyclic data-flow graphs of stream-oriented operators (e.g., [7]), operating over the global schema. XFlow has a built-in set of standard windowed operators (filters, unions, aggregates, joins) and also allows for arbitrary user-defined functions to be linked as operators. An example of three simple queries is shown in Figure 5.1(b). When a client subscribes a query,

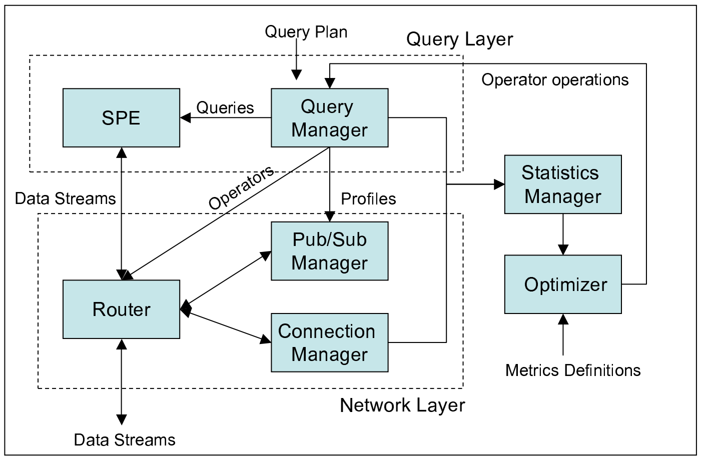


Figure 5.3: XFlow node architecture.

the operators of the query get subscribed to their input streams. This is done along the upstream to downstream direction, in an order that is consistent with a topological sort of the operators in the query plan.

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

Figure 5.1(b) and 5.2 shows three queries and a possible distribution of their operators across the network. Q_1's operators are distributed across two trees. F_1's host, n_1, joins tree T_1, to receive stream S_A, and publishes its output to tree T_2. Hence, n_4, the host of A_2, joins T_2. Node n_5, the host of Q_2, joins also T_2. Moreover, it joins T_3 and subscribes to stream S_B. Finally n_6, hosting F_4, joins tree T_3.

5.1.5 Node architecture.

The architecture of an XFlow node is shown in Figure 5.3 and reveals how we integrated the pub-sub model with a stream processing system (SPE). Query plans are parsed by the *query manager* and sent to the SPE for execution. Profiles are extracted and pushed to the *pub/sub manager* which contacts the bootstrap and requests the trees publishing the streams specified by the profile. Nodes join trees through the *connection manager* which maintains the connections with each node's parents and children. The *router* delivers input streams to the SPE and output streams to clients or downstream nodes in the trees. It also routes any maintenance data required by the connection manager as well as any profile updates sent by the pub/sub manager. The *optimizer* is the extensible component in our architecture. Application designers can specify their own performance criterion (see details in Section 5.3), and XFlow customizes its functionality based on these metrics. Using data on the query and the overlay network, obtained from the *statistics manager*, it identifies operations on the queries that improve the cost metrics and sends any query changes to the query manager.

5.1.6 Benefits of our model

A distinguishing feature of XFlow is its support for dynamic creation of multiple overlapping dissemination trees and for optimizations across multiple trees. In [37, 44] it was shown that multiple trees enable better network utilization and reduce redundant transmissions by having clients become part of only those trees that publish relevant data. Furthermore, XFlow treats query operators as stream subscribers and publishers. In contrast to existing systems, i.e., Borealis [6], our model decouples operators, i.e., an operator does not require any state regarding the location of its upstream/downstream operators in the query network. This allows XFlow to handle a large the number of dynamic operator changes as changes do not require updating any state regarding their neighbors in the query network. Moreover, XFlow goes beyond centralized implementations of the pub-sub semantics, as in [34, 39, 45], and realizes this model through *dynamic* creation of multiple network trees.

5.2 Tree management.

In principle, XFlow creates one tree per each source (internal or external). This basic approach may seem unwieldy, as each source and operator will have its own tree. In practice, we employ simple techniques to control the number of trees. First, we allow the definition of super-operators that combine multiple connected operators in a single unit. A single tree is created for each super-operator, as the constituent operators will not publish data. Moreover, we expect that many overlay trees will be entirely local to a single node at the network level; for example, in the case when an operator and its upstream parent operator are located on the same node.

Second, sources can be assigned to already existing trees, reducing the number of trees. For example, streams requested by highly overlapping sets of subscribers can be published through the same tree, while streams can be periodically reassigned across root brokers, adapting to membership changes. A simple heuristic could be to discover, using bloom filters, trees that involve the same set of nodes and merge them. The problem of effectively grouping data sources and mapping them to a given set of trees has been studied in [8, 44] and is beyond the scope of this paper. For simplicity of exposition, we assume that this grouping is already done and use source to mean collections of

```
01. system cost:= aggregate (aggregation function, node cost, BROKERS) |
                  aggregate (aggregation function, query cost, QUERIES)
02. node cost:= aggregate (aggregation function, operator cost, LOCAL OPERATORS)|
                aggregate (aggregation function, local stats)
05. query cost:= aggregate (aggregation function, operator cost, QUERY OPERATORS)|
                aggregate (aggregation function, local stats)
06. operator cost:= aggregate (aggregation function, local stats, operator set)
07. operator set:=UPSTREAM OPERATORS|DOWNSTREAM OPERATORS
09. local stats:= path latency|incoming data rate|operator selectivity, node load...
10. constrained metric:=(constraint metric, operator, threshold)
11. constrained metric:= system cost|node cost|query cost|operator cost
12. operator:= < | > | ≤ | ≥ | ≠
13. aggregation function:= MIN|MAX|SUM|AVERAGE
```

Figure 5.4: XFlow grammar

sources.

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

5.3 Cost definition model

The cost model of XFlow extends our basic cost model defined in Chapter 3 along two non-trivial dimensions. First, it incorporates high-level metrics that can express the efficiency of user queries as well the processing overhead on the network nodes. Finally, we express and optimize metrics and constraints across a general network of *multiple* trees,, whereas our basic model addresses the case of a single tree. Based on our model, XFlow can be customized for different optimization metrics. In contrast to existing solutions [9, 45] that focus on specific metrics, our cost model can express a variety of performance measures through a sequence of statistics aggregation steps. We note here that generic cost models where also proposed in [39]. However, they use a less expressive model that simply sums network link costs. A summary of our model is shown in Figure 5.4. In this section, we describe XFlow's cost model in detail.

5.3.1 Optimization metrics and constraints

Distributed stream processing systems are often evaluated by query-related QoS targets [6] as well as resource utilization metrics [9, 45]. The former refers to performance of queries (e.g., response latency), whereas the latter addresses the in-network processing overhead (e.g., bandwidth consumption). Our cost model is designed to express both metric types.

To facilitate the expression of a broad spectrum of metrics, nodes evaluate a set of built-in *local statistics* for the overlay links they maintain, e.g., link latency/bandwidth capacity, as well as their local operators, e.g., input/output rates, selectivity, processing costs. Application designers can combine these statistics and define optimization metrics and constraints.

QoS metrics XFlow expresses the performance of a query based on the performance of its operators. Specifically, we define the *query cost* as an aggregation of its *operator costs*, where the *system cost* is an aggregation of the query costs. Applications can use the aggregation functions shown in Figure 5.4 and customize XFlow for various metrics, e.g., average query output latency, query load, query throughput.

XFlow users may define the operator cost based on (i) local statistics or (ii) the aggregation of network statistics. The former case captures metrics like the operator's processing latency or its load. The latter expresses network-related metrics, such as response latency, which includes the dissemination delay for fetching the input stream from its producer plus its processing latency. Nodes aggregate statistics of the network links connecting an operator to its neighbors in the query plan, that is the location of (i) its upstream operator or (ii) its immediate downstream operator. If there exist multiple upstream or downstream operators, we define the cost of each one independently and average the costs.

The maximum query latency is an example of upstream aggregation: an operator's latency is the sum of the link latencies on the path to its upstream operator plus its processing latency. A query's latency is the sum of its operators' latency (QUERY OPERATORS).

```
operator latency= aggregate(SUM, latency, UPSTREAM OPERATOR) + processing latency
query latency= aggregate(SUM, operator latency, QUERY OPERATORS)
system cost= aggregate(MAX, query latency, QUERIES)
```

Resource utilization metrics XFlow allows also the definition of metrics that measure the resource utilization of nodes. We refer to such metrics as the *node cost* and can be defined based on

(i) the local statistics or (ii) the aggregation the local operators' cost. In this case the system cost is the aggregation of the node costs. An example is the processing load of a node, which is the total load of its locally executed operators (LOCAL OPERATORS). Based on this, we define below the system cost as the maximum processing load across all nodes:

node load= aggregate(SUM, operator processing load, LOCAL OPERATORS)
system cost= aggregate(MAX, node load, BROKERS)

Another example is the outgoing bandwidth consumption of a node. This is the data the local operators publish (*outgoing rate*) plus the data it forwards in the network as part of the dissemination trees. Hence, the total bandwidth consumption across all brokers is expressed as:

node out rate	= aggregate(SUM, operator output rate, LOCAL OPERATORS)
	+ forwarding rate
system cost	= aggregate(SUM, node out rate, BROKERS)

Constraints Using our cost model, applications can express constraints for queries-related and resource utilization metrics. They can define cost metrics for operators, queries or nodes and specify bounds on them, e.g., maximum node load, maximum query response latency, etc. For example, they can express constraints to guarantee sufficient bandwidth capacity on the overlay links connecting an operator with its input producer. This is crucial for applications with high input rates. The capacity between two operators is the minimum available bandwidth of the network links connecting them:

operator capacity	= aggregate(MIN, bandwidth, UPSTREAM OPERATORS)
operator capacity	$\geq$ operator input rate

5.3.2 Statistics collection

XFlow nodes are customized to collect and aggregate statistics required for the evaluation of the performance metrics and constraints. To reduce the statistics traffic, nodes evaluate partial results and collaborate in order to derive the final cost metrics. Specifically, depending on the definition for operator cost, (i.e., upstream/downstream aggregation), each node on the path connecting two operators evaluates the required local statistic on the link to its parent/children in the dissemination tree and aggregates this with the aggregated metric of its parent/children respectively to derive its final cost value. For example, to measure the latency to the root of a tree, a node measures the link latency to its parent and adds that to the latency of the parent to the root (which it receives periodically from its parent).

Based on this aggregation model, each node n_i maintains *only two* statistical values for every dissemination tree in which it participates:

- the local statistic value, l_i, (e.g., latency to parent) and,
- the aggregation of network statistics, ϕ_i (e.g., latency to root).

Hence, our cost model requires state of *constant size* per node, which considerably improves the scalability of the system.

5.4 Distributed optimization

As mentioned in Chapter 3, the goal of the optimization process is to minimize the application-specified system cost and, through a set of optimization rules, adapt to time-varying network or workload conditions. In XPORT these rules modify the in-network distribution and implementation of the subscribed stream-based queries. We refer to these rules as *query-based optimization rules*. Its unique characteristic is that it employs a metric-independent model that dynamically combines operator migration, replication, and partition and can uniformly handle a variety of cost metrics. Our framework does not rely on global information and has low communication overhead. Each node maintains aggregated information for its own "neighborhood" and periodically distributes its operators across its neighbors. We refer to these operations as *localized optimizations*.

To avoid local optimum, we also utilized non-local optimizations, following the basic model described in Chapter 3. Hence, we selectively distribute certain aggregated statistics to any potentially interested nodes and allow them to consider specific *directed* optimizations. These operations consider only network nodes that demonstrate a good performance and their resources could be used to improve the system performance. In the rest of the section, we describe our optimization framework in detail.

5.4.1 Query optimization rules

Nodes distribute their operators across their neighbors in the trees, i.e., its parents and its descendants up to k-levels in each tree, where k is a system parameter. We denote as n_i^k a node n_i that is included in tree T_k, and Figure 5.5 shows the neighborhoods of n_i in two trees, T_1 and T_2.

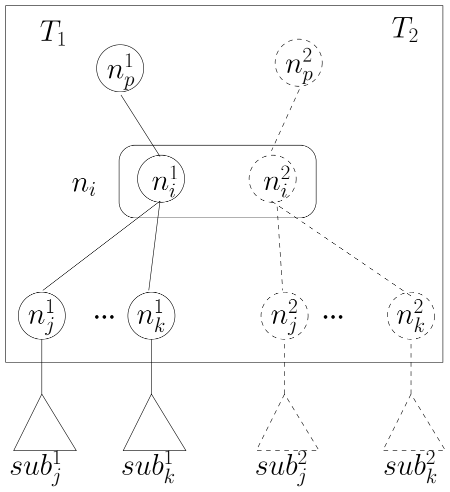


Figure 5.5: Optimization area of n_i in XFlow

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

Operator placement and migration

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

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

Operator replication and partitioning

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

Replication Replication is applied to operators that subscribe to multiple trees. Instead of collecting all streams and executing the operator on a single node, replication exploits processing resources of multiple trees. The goal is to process each input stream independently on each tree and combining the results to construct the final output. To implement this, a replica is created for each of the trees the operator receives input from. Each replica will receive one of the inputs of the original operator. To guarantee correctness, a *final operator*, receives the outputs of all replicas and produces the final result. The publish-subscribe network easily routes stream flows from the replicas: they publish their outputs through a tree, and the final operator's host subscribes to them by joining these trees.

Depending on the semantics of the replicated operator, we need to use a different implementation for the final operator. For example, for filters we merge the already filtered outputs of our replicas using a union operator, while for count operators, the final operator sums the replicas' results.

In our current implementation we used feed processing operators similar to the operator set in Yahoo!Pipes [5]. The set of operators along with the operators used for their replicas and final operators are shown in Table 5.1.

Operator	Replica Operator	Final Operator
count	count	sum
filter	filter	union
rename	rename	union
copy	copy	union
split	split	union
truncate	truncate	union
union	union	union
unique	unique	unique

Table 5.1: Implementation of operator replication and partition.

Replication for the count operator is shown in Figure 5.6. A union operator precedes the count merging streams A_1 and A_2 into stream A. We count the attribute x, thus, after replication, we apply the count independently on A_1 and A_2 and carry the results in the c attribute of their replicas' outputs A'_1 and A'_2. We sum the c value from both streams (after we merge them) in order to get the final result.

The benefits of replication primarily depend on the location of the replicas. For example, if improving the query latency is our objective, then placing the replicas in Figure 5.6 on the same node as the original operator will not decrease latency. To increase the benefit, nodes can migrate

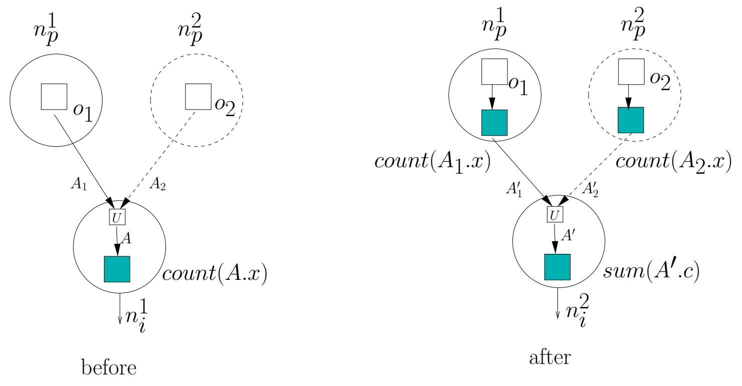


Figure 5.6: Operator replication.

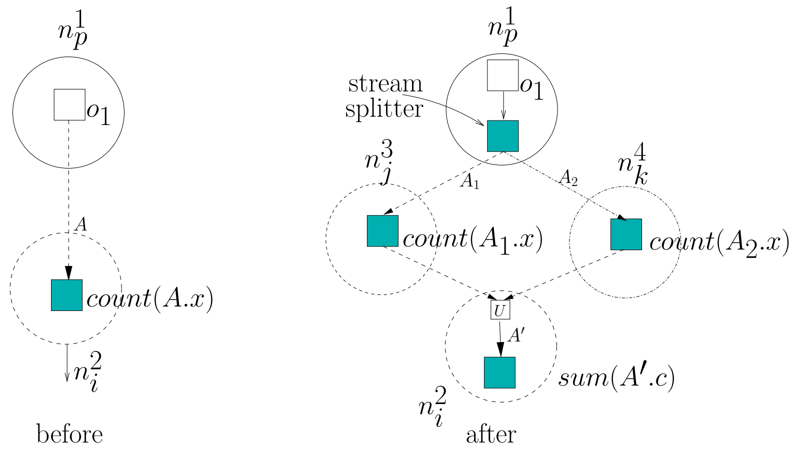


Figure 5.7: Operator partition.

the replicas inside their optimization area. Since each replica corresponds to a different tree, a node can place the replica on its neighbors on this tree, i.e., its parent, one of its children or siblings. Note that we do not require *all* replicas of an operator to be migrated, only the ones that can further improve performance.

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

Operator Partition Partition is applied on the input stream of an operator. It splits the stream into two flows and each sub-flow is processed by a different replica of the operator, thereby exploiting data parallelism. The output of the replicas are processed by a final operator which produces the final results. Its type depends on the original operator, similarly to the case of replication. Replicas created due to stream partitioning can be migrated inside the optimization area. The hosts of the replicas join the trees providing their inputs and the publish-subscribe network routes the output streams published by the replicas to the location of the final operator.

Partition is shown in Figure 5.7 for the count operator. Node n_p^1 splits stream A and publishes two sub-streams A_1 and A_2. These streams are routed to the two replicas and their outputs are merged and processed by the final operator. For this example, we assume that node n_i participates in trees T_3 and T_4 and it places the replicas on some instances of its neighbors n_j and n_k on these trees.

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

5.4.2 Local optimizations

XFlow applies the above optimization operations in the scope of its *local* neighborhood. Hence, nodes may migrate their operators (or replicas) to their parents, children or siblings across trees. The main advantage of this approach is that we reduce the search space of candidate locations for an operator by considering only nodes within its network neighborhood. Furthermore, non overlapping neighborhoods can be optimized locally and concurrently.

Although at each step we consider a small network region, XFlow can gradually migrate operators to arbitrary network areas. Our experimental results reveal that for certain types of metrics this local search is scalable and effective, as it incurs very small network traffic and can converge to near-optimal configurations. For example, for non-constrained additive metrics, like average network latency, or total bandwidth consumption, our optimizations can migrate all queries close to their external sources.

5.4.3 Directed optimizations

XFlow utilizes the available network statistics and allows nodes to identify promising non-local operations. These operations focus on specific low cost network areas and we refer to them as *directed* optimizations. In this section, we describe our approach.

Statistics management

XFlow nodes maintain certain local and aggregated statistics which they *selectively* disseminate to discover alternative neighborhoods that could improve the performance. XFlow relies on the definition of the system cost metrics, i.e., on the semantics of the aggregation functions (i.e., MAX, SUM, etc), in order to determine: (i) which nodes may be interested in the statistics, and (ii) which statistics could be of potential interest. Moreover, it creates filter-based routing paths on top of the existing overlay trees that forward the statistics from their producers only to the interested consumers. This statistics propagation process should run with a frequency that reflects the workload and network changes of the specific application.

Statistics selection In Section 5.3.2, we mentioned that each node n_i stores a local value, l_i, and an aggregated value, ϕ_i, for every tree in which it participates. It periodically disseminates these statistics and informs its peers about the performance of its neighborhood. Specifically, the local value l_i reveals n_i's resource utilization (e.g., processing load) or its neighbors' properties (e.g., latency to its parent), while the aggregated value ϕ_i provides a performance measure of the *path* leading to n_i (or of the links to its children). Each node forwards statistics of constant size ($O(1)$) for each tree is participates. The size of these statistics is independent of the number of queries and operators as well as the number of nodes and only depend on the number of dissemination trees.

Selective dissemination XFlow distributes statistics only to nodes that may affect the global system cost. Table 5.2 shows the conditions that should hold in order for a node to receive any statistics from its peers. These conditions are *agnostic* of the actual optimization metric and depend on the aggregation functions that evaluate the query and system cost. For example, if the optimization metric is the average query latency, then every node can improve the system cost by reducing the processing or network delay of all the queries it hosts. On the other hand, in order to minimize the maximum query latency, we would forward statistics only to the nodes that process the query with the highest response time, c, (or with small difference $\pm\delta$ from the worst response time).

Furthermore, nodes are interested only in information on network components (links, paths, nodes) that perform better than their own local neighborhood. Table 5.2 shows the predicates used by each node n_i to filter out non-informative statistics. XFlow uses the aggregation semantics to *automatically* customize these filters. Additive functions imply that the aggregated value will be higher than the local value, thus the lower-bound filter is their local value. For example, for the query latency case, n_i receives statistics about paths with less latency than its latency to its parent, l_i. In the case of the MIN function, the aggregated metric ϕ_i provides the lowest value n_i is interested

Operator Aggr	**Disseminated Statistics**	**Statistics Filters**	**Query Aggr**	**System Aggr**	**Condition**
SUM	$(l_i,\ \phi_i)$	$< l_i$	SUM	SUM	
			MIN	MIN	$(\phi_i = c \pm \delta)$
MIN	(l_i, ϕ_i)	$> \phi_i$	SUM	SUM	
			MIN	MIN	$(\phi_i = c \pm \delta)$
-	l_i	$< l_i$	SUM	-	
			MAX	-	$(l_i = c \pm \delta)$

Table 5.2: Statistics dissemination per node. (c: system cost)

in; n_i will receive statistics about links and paths with higher bandwidth than the capacity of its own local path to the root, ϕ_i.

Probabilistic dissemination In order to reduce the statistics emitted in the network, we deploy a probabilistic-based technique for selecting only a subset of the available information. Specifically each node propagates the statistic of k nodes (we refer to them as *top-k*) which are picket based on the *lottery scheduling* algorithm [53]. Each node's value is assigned a number of tickets proportional to its "utility". This utility value depends on the semantic of the optimization metric. For example, nodes with less load or latency will be given more tickets, thus having higher probability of "winning" the lottery and forwarding their statistics. The algorithm guarantees no zero probability for selecting any statistic. Our experiments revealed that this approach keeps the statistic traffic within constant bounds, independently of the number of queries and nodes, while it incurs low performance degradation.

Statistics routing XFlow uses the structure of its network to distribute its statistics. It avoids flooding the network by constructing predicate-based routing paths that filter out unwanted statistics as early as possible. For example, we avoid forwarding load information of the most loaded node. To construct these filtering paths, nodes propagate their statistics filters (Table 5.2) to their parents. Each node aggregates the filters of its children in each tree and propagates the aggregated filter upstream towards the root. This allows nodes to be aware of the interests of their descendants and selectively route filtered statistics to the interested nodes. Since our overlay network consists of inter-connected trees, every participating node is guaranteed to receive the statistics.

Directed operations

Our framework uses the collected statistics to discover specific low-cost neighborhoods. We categorize them into *intra-tree* and *inter-tree* neighborhoods.

Intra-tree neighborhoods Dissemination trees connect nodes that receive and process the same data stream. Our statistics distribution process allows nodes to discover which of these nodes have better cost metrics, e.g., better path to the stream producer, less processing load, or less outgoing bandwidth consumption. Hence, each node considers migrating its operators (or its replicas) to a peer that receives its input stream through an alternative path with better performance. Another benefit of directed optimizations inside a tree is that overlay paths with good performance are re-utilized, improving the resource utilization metrics.

Inter-tree neighborhoods XFlow nodes also receive statistics regarding nodes, links and paths that reside outside their dissemination trees. Thus, they can exploit any network component with low cost by incorporating them in their own trees. To achieve this, they consider migration of their operators to any node with lower cost and connect them to the tree through one of the existing nodes. To reduce the number of candidate locations each node has to consider, we check only the top-k nodes with the least cost.

5.4.4 Evaluating global cost changes

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

We start by providing the dependencies among nodes, operators and queries. We assume a set of queries Q. Each query $q_i \in Q$ consists of a set of operators and let O be the total set of operators. Let also E_i be the set of queries that include operator $o_i \in O$. We denote the local metric of node n_i as l_i. Any optimization rule that involves this node could affect its local metric (e.g., adding an operator increases its processing load). We denote such changes as $\Delta(n_i : l_i \rightarrow l_i + \delta)$. The cost of an operator $o_i \in O$, oc_i, is defined by aggregating the local metrics of some nodes (e.g., latency of the nodes to the upstream operator). This cost could change due to changes on the local metrics (e.g., increase on the link latency) or changes on the set of nodes (e.g., placing the operator on another node.) The following definition identifies which non-local operators and queries (that do not reside on this node) may be affected by a change on a node's local metric.

Definition 6. (DEPENDENT OPERATORS) *The* dependent operators *of node n_i is the set of operators, D_i, whose cost metrics depend on n_i's local metric, l_i:*

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

Table 5.3: Model terminology

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

2. *If the cost of an operator is defined as a downstream aggregation, then D_i is the set of first operators reached by n_i's path to the root of every tree n_i participates.*

Definition 7. *The* dependent queries *of n_i is the set of queries, G_i, that include at least one dependent operator of n_i. In particular: $G_i = \cap_{o_j \in D_i} E_j$.*

Let us consider the query plan and its distribution in Figure 5.8, where the cost of an operator is its latency (i.e., upstream aggregation). Then $D_2 = \{o_4\}$, as changes on the latency between n_2 and n_3, affects the output latency of o_4.

Each optimization rule α affects a set of nodes F_α: migration has an impact on the origin and destination nodes, while replication and partition affect the origin node and the nodes where the replicas are placed. Changes on a node $n_i \in F_\alpha$ could change the cost of all dependent queries as well as the cost of queries that include any operator executed on these nodes. Let O_i be the set of operators executed on n_i and Z_i the set of queries including an operator in O_i. XFlow does not evaluate the new cost of every dependent query, but instead quantifies the impact of their cost changes on the global system cost, expressed by the *query dependence cost*.

Definition 8. (QUERY DEPENDENCE COST) *The* query dependence cost*, $c(G_i)$, of a query dependent set G_i, is the aggregation of the cost of every query $q_j \in G_i$, using the aggregation function that defines the system cost.*

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

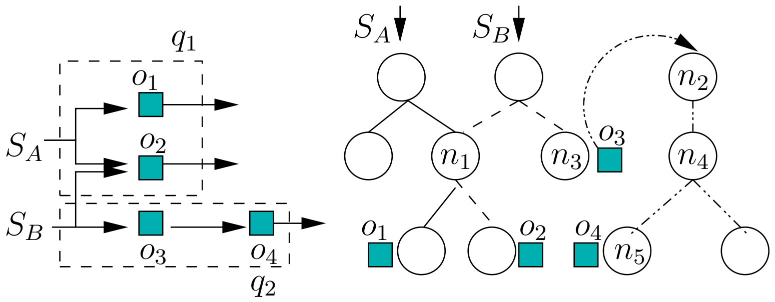


Figure 5.8: Example of operator distribution.

$$c(G_i) = \sum_{q_m \in G_i} qc_m = \sum_{o_j \in D_i} \sum_{m \in E_j} qc_m. \tag{5.1}$$

Our cost model tries to estimate the change on the query dependence cost. In what follows, we illustrate the detail of this approach. The following property identifies the effect of an operation on the system cost, based on the query dependence cost.

Property 1. (SYSTEM COST EFFECT) *Assume the system cost is defined based on the* SUM *function. Given a query optimization rule α, the expected change of the system's performance is the following:*

$$b_\alpha = \sum_{n_i \in F_\alpha} \Big(\sum_{q_m \in Z_i} \Delta qc_m + \Delta c(G_i) \Big). \tag{5.2}$$

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

$$\begin{aligned} b_\alpha \quad = \quad & \min_{n_i \in F_\alpha} \{ \min_{q_m \in Z_i} \{qc_m + \Delta qc_m\}, c(G_i) + \Delta c(G_i), \\ & \min_{q_m \notin Z_i, q_m \notin G_i} \{qc_m\}\} - c \end{aligned} \tag{5.3}$$

where c is the current system cost.

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

Query cost changes

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

Definition 9. (CRITICAL VALUE) *Let n_i be a node. If the cost of operators is defined based on the* MIN *function, then the* critical value *of a dependent operator $o_j \in D_i$, w.r.t. n_i, $h_i(o_j)$, is the minimum local metric of all nodes on the path between n_i and the current location of o_j.*

To explain this, we use the example in Figure 5.8, and let us assume that the cost of an operator is the minimum latency to its upstream operator. This is an upstream aggregation, thus n_2 has o_4 as its dependent. The local metric of each node is the latency to its parent and $h_2(o_4)$ is the minimum latency of all nodes connecting n_2 and n_5.

Property 2. (OPERATOR COST EFFECT) *Assume a change $\Delta(n_i : l_i \rightarrow l_i + \delta)$ that triggers a change Δoc_j on the cost of a dependent operator $o_j \in D_i$. If the operator cost is defined based on the* SUM *function, then $\Delta oc_j = \delta$, while if it is defined based on the* MIN *function, then $\Delta oc_j = \lambda$, where $\lambda = \min\{l_i + \delta, h_i(j)\} - \min\{l_i, h_i(j)\}$.*

To explain the above, we assume in Figure 5.8 that the operator's cost is its output latency. Hence, we aggregate the link latencies between nodes n_5 and n_3 to get the latency of operator o_4. If the latency between n_4 and n_2 increases, then the latency of o_4 will have the same increase. If the operator cost is the minimum latency to its upstream operators, then the cost of o_4 is the minimum latency of the links between n_5 and n_3. A change on the latency between n_4 and n_2, will change the cost of o_4 by λ. The λ parameter identifies the difference between the new minimum latency between nodes n_5 and n_3 and their current latency. This parameter can be further simplified under certain conditions as shown in Table 5.4. For instance, if the node with the minimum latency is the same node for all the dependent operators, and we only decrease further its latency, then every dependent operator will experience the same cost change.

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

Property 3. (QUERY COST-EFFECT) *Assume a change $\Delta(o_i : oc_i \rightarrow oc_i + \delta)$ that triggers a change Δqc_j on the cost of a query $q_j \in E_i$. If the cost of a query is defined based on the* SUM *function, then $\Delta qc_j = \delta$ and if is defined based on the* MIN *function, then $\Delta qc_j = \tau$, where $\tau = \min\{\beta_i(j), (oc_i + \delta)\} - qc_j$ and $\beta_i(j) = \min_{o_m \in P_j, o_m \neq o_i}\{oc_m\}$.*

Assume the query cost is the sum of its operators' latencies. Then, in Figure 5.8, decreasing the latency of o_2 will decrease the cost of q_1 and q_2. If the query cost is defined as the minimum

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

Table 5.4: Change on operator cost oc_j upon change on local metric l_i (oc_j uses the MIN aggregation function).

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

Benefit evaluation steps

The above properties allow us to estimate the benefit of a given query optimization rule. First, we evaluate the change on the cost of the migrated operator (or replica) by aggregating the local metrics for its potential new neighbors. The aggregation function and the set of neighbors depend on the definition of the operator cost (upstream or downstream aggregation). Given a change on the operator cost, we quantify the impact on their query costs based on the query effect property.

In the next step, we estimate any changes on the local values of the affected nodes, based on the definition of the local metric. Since the local metric is a function on local statistics, we evaluate the new local metric based on the local statistics of the affected nodes. For example, if the local metric is the node's processing load, and we migrated an operator to the node, then we add on the destination node's load the expected processing load of that operator, as estimated by its current host's statistics. Depending on these changes, our protocol uses the operator and query cost-effect properties to evaluate the impact on the dependent queries of these nodes and the new query dependence cost. Based on this impact, we use Equation 5.2 or 5.3 to evaluate the benefit of the query optimization rule.

5.4.5 Query optimization protocol

Periodically, every node evaluates the benefit of all query optimization rules on its local operators. For each operator, it considers all possible migrations, replications and partitions (the last two operations are combined with the possible replica migrations). The most effective of all pairs (operator, optimization rule) is sent to the root of the tree. The roots of all trees collaboratively identify the best rule which is applied by the host of the operator.

Oper Aggr	**SUM**		**MIN**	
Query Aggr	$\|Z_i\|$	$O(\|Q\|)$	$h_i(o_j), \forall o_j \in O_i$	$O(\|O\|)$
SUM	$\|G_i\|$	$O(\|Q\|)$	$h_i(o_j), \forall o_j \in D_i$	$O(\|O\|)$
			$\|E_j\|, \forall o_j \in D_i$	$O(\|O\|)$
			$c(G_i)$	$O(1)$
MIN	system cost	$O(1)$	system cost	$O(1)$
	$c(G_i)$	$O(1)$	$c(G_i)$	$O(1)$
	$\beta_i(j), \forall q_j \in Z_i$	$O(\|Q\|)$	$h_i(o_j), \forall o_j \in O_i$	$O(\|O\|)$
	$B_j, \forall o_j \in D_i$	$O(\|O\|)$	$\beta_i(j), \forall q_j \in Z_i$	$O(\|Q\|)$
			$h_i(o_j), \forall o_j \in D_i$	$O(\|O\|)$
			$B_j, \forall o_j \in D_i$	$O(\|O\|)$

Table 5.5: Optimization state for n_i ($B_j = \{(\beta_j(m), oc_j) | q_m \in E_j\}$).

Dynamic operator modifications

Our system adopts the "pause-drain-resume" approach to migrate or change the execution of stateless operators. When a node decides to modify an operator, its pauses the data flow to that operator, and starts buffering any incoming tuples. The operator executes any remaining tuples and after the operation is applied the node resumes the data flow. To handle migration of stateful operators we adopt solutions proposed in the literature [56].

Optimization state

Table 5.5 shows the state nodes need to maintain for the different functions for aggregating the operator and query costs. This state is derived based on Equations 5.2 and 5.3 and the operator and query cost-effect properties. For example, to use the operator effect property for the MIN function, nodes need to know the critical value of their local and dependent operators. By maintaining this state we reduce the communication overhead during the optimization process. Note that this state is common for any query optimization rule and independent of the actual performance metric. Moreover, it depends in most cases on the dependent entities of a node and not on the global set of nodes, queries or operators in the system.

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

Optimization traffic

During optimization, nodes exchange statistics with their neighbors in the optimization unit. This is the information required from our protocol to evaluate the changes on the local metrics of the affected nodes and on the operators to be migrated, replicated or partitioned. This state depends on the definition of our cost metrics. For example, for the query latency metrics defined in Section 5.3, nodes need to know the latency of the candidate locations from the upstream operators. Thus, they collect the local metrics from the nodes connecting the candidate location to the location of the upstream operators.

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

Batch-executing multiple optimization steps

Although applying a single query optimization rule per optimization period will improve performance over time, considering multiple rules could speed up convergence to better configurations. To this end, we allow multiple rules to be applied during a single period, e.g., migrating multiple operators.

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

5.5 Performance evaluation

We have implemented an initial prototype of XFlow in Java and studied its performance on the PlanetLab testbed. We used a network of up to 200 nodes participating in four dissemination trees. Thus, our networks consist of up to 800 node instances. The workload includes 900 queries with input streams chosen from a set of 700 real-world input streams using a Zipf distribution with the skew parameter set to 0.97. Each input stream is an RSS feed pulled with frequency that creates an average stream rate of 3.2KB/sec.

Operator	Description
count	counts the number of items in a feed
filter	blocks items that match a filtering rule
rename	renames item attributes
copy	copies item attributes
sort	sorts items based on an ordering rule
split	splits a feed into two identical copies
truncate	limits the number of items that pass through it
union	merges two feeds together
unique	combines items containing identical strings

Table 5.6: RSS feed processing operators.

Our queries are composed by a set of operators, similar to the operators of Yahoo!Pipes [5], a centralized feed aggregator that lets users mashup data sources. The operators may union, split, sort or filter the RSS feeds with a variety of conditions. The operators used are described in Table 5.6. Each query takes as input a random RSS feed (or two for the union operator) and applies a chain of five processing operators. An example query unions two input feeds, filters them based on a string in the title, sorts the results by date, truncates them and returns the top-k items.

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

We used our prototype to implement three query distributions, each one optimizing a different metric which are: (i) average query latency, (ii) maximum processing load across all nodes and (iii) total bandwidth consumption. These metrics were defined in Section 5.3. Our experiments demonstrate XFlow's effectiveness, as it manages to improve these metrics significantly over a sequence of local and directed optimization rules.

5.5.1 Extensibility and effectiveness

We start our discussion by demonstrating that our operator placement framework can optimize efficiently different metrics. We examined four alternative placement approaches.

- *CLIENT-STAR* assigns operators to the location of their client and their hosts connect directly to the root brokers, creating a star topology for every dissemination tree.

- *CLIENT-TREE* assigns the operator to the host of the client, however, these hosts connect to the trees through a random member of the tree. CLIENT-STAR is also the initial configuration for XFlow.

- *SOURCE* places operators to the root brokers publishing their input stream. The root brokers process all the queries what have as input their publishing streams and forward the results to the clients through direct, unicast connections to the client hosts.
- *GLOBAL* applies a greedy strategy that considers all existing queries in the order they were registered to the system and places each after an exhaustive search over all possible placements. This requires global knowledge of the query set and workload and is infeasible in practice but it gives a target upper bound for the performance of our algorithms. For each metric, we used a different implementation of GLOBAL.

We compare these approaches with two variations of XFlow:

- *XFlow-local* uses only localized optimization rules
- *XFlow-directed* uses only directed optimization rules and
- *XFlow-hybrid* uses both localized and directed rules

Our results show that, although the best placement depends on the optimization metric, XFlow consistently performed very close to the best placement *regardless* of the performance metric. Furthermore, we show that certain metrics can converge to the optimal placement by using only localized operations, while for the rest of the metrics, directed optimizations allow XFlow to achieve the optimal configuration or reach one that performs better than GLOBAL.

Bandwidth consumption

Figure 5.9 shows the total bandwidth consumption for varying number of queries and 100 nodes. In these experiments each query is a chain of five filter operators for which we manually set the selectivity to be uniformly distributed in [0,1]. The best performance in this case is achieved by placing operators with selectivity less than one close to the sources. This approach eliminates input tuples close to their sources reducing the amount of data forwarded in the network. Thus, both SOURCE and GLOBAL place the operators on the root brokers and represent the *optimal* placement. CLIENT-STAR consumes more bandwidth since all input tuples are forwarded to the clients for processing. CLIENT-TREE performs worse, because input tuples are forwarded to their clients through multiple hops. However, XFlow manages to continuously refine the operator placement by using only its local optimizations (*XFlow-local*). It converges to a distribution that requires low

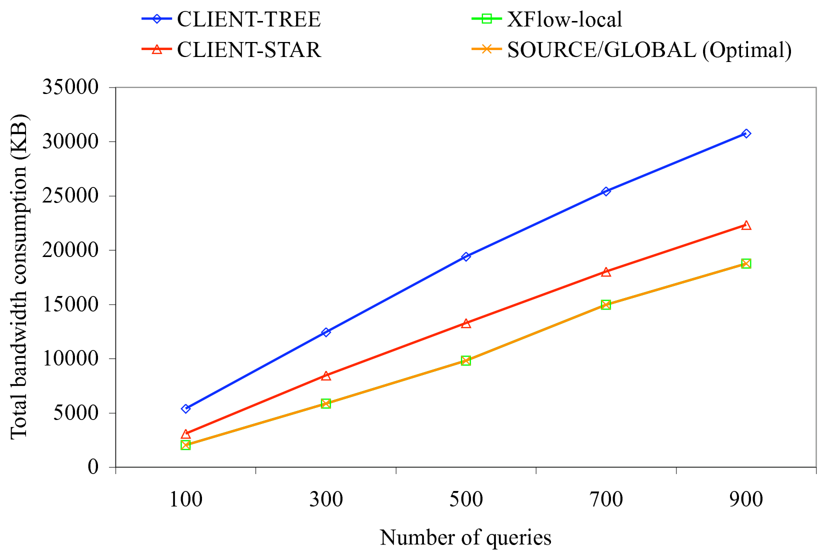


Figure 5.9: Convergence to the optimal for the total bandwidth consumption.

bandwidth consumption, i.e., over time, it moves almost all operators to the root brokers, performing the same as GLOBAL.

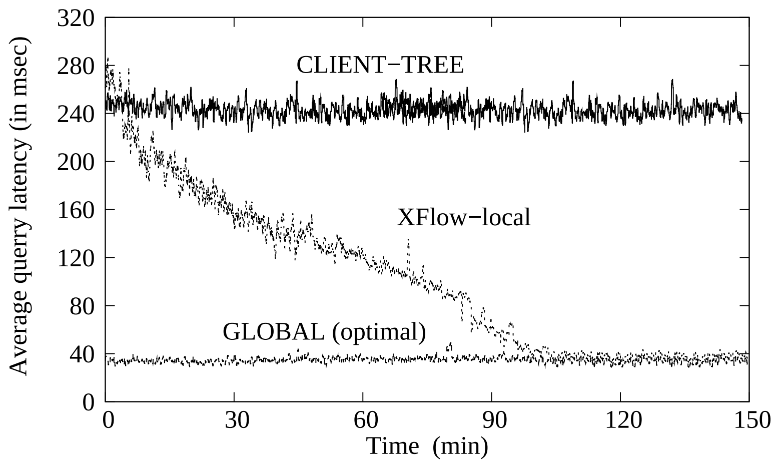


Figure 5.10: Convergence to the optimal for the average query latency.

Query latency

Figure 5.10 shows the average query output latency for 500 queries deployed on 100 PlanetLab sites. The query response latency includes only the network latency incurred due to the operators' placement on different network location. In this case, network latency refers to the dissemination time for distributing streams from producers (external sources or internal operators) to consumers (external clients or internal operators).

We compare XFlow's performance with the *optimal* in which each node has one overlay link to the root broker that publishes the input stream of its operators. This is provided by the GLOBAL as well the CLIENT-STAR placements. In these cases, assuming network latency dominates processing latency and no constraints on the node load, we execute each query on the root and forward the results the client. This reduces the query latency as input tuples do not travel in the network. CLIENT-TREE uses multiple overlay links to connect an operator to its source, hence, its performance is worse. Figure 5.10 shows that XFlow converges very close to the optimal configuration after a number of migrations, which incrementally move our operators closer to the root of each tree.

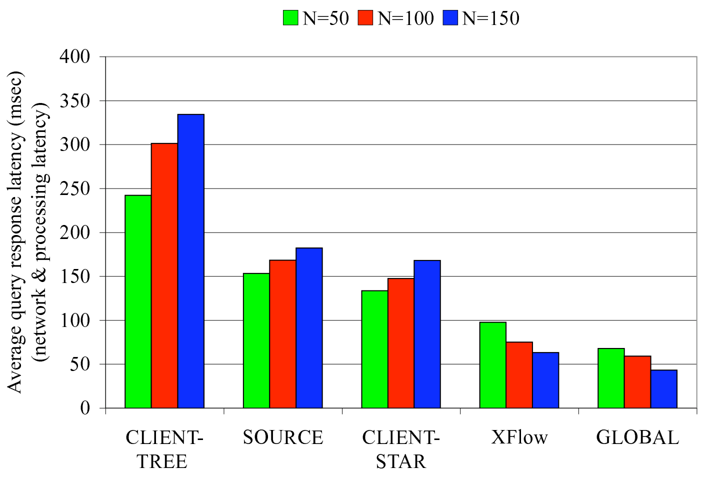


Figure 5.11: Average network and processing latency.

We also studied the query response latency when processing times are also included in the evaluation of the query response latency. The results for different network sizes are shown in Figure 5.11 for 500 queries. The more operators a node must process and the higher its fanout in the tree, the greater the processing latency that feeds observe when traveling through that node. For the GLOBAL case, we assign operators to the node with the lowest latency from the root broker publishing their input. To reduce further the network latency, we assign clients to the same node as their queries, so that output results are not forwarded in the network.

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

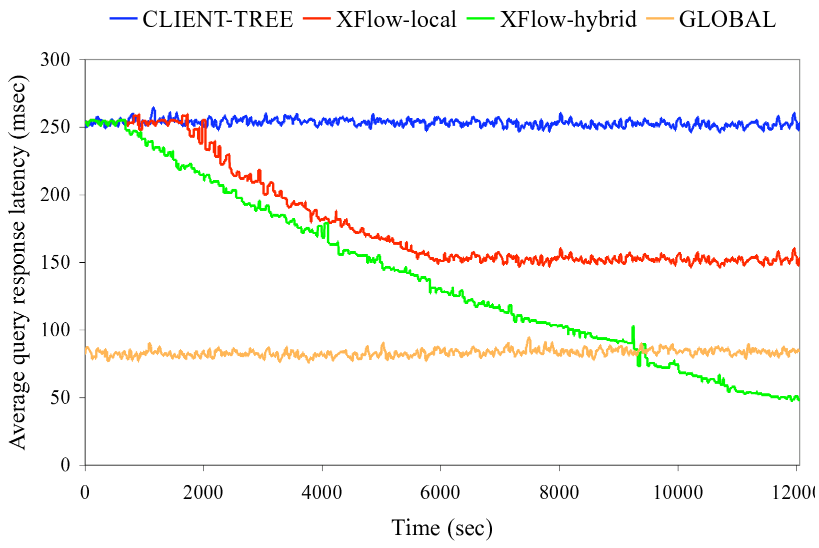


Figure 5.12: Average query response latency when maximum node load is restricted to five (5) CPU cycles per time unit

of the roots, and thus the processing latency, increases. SOURCE has worse performance, since the roots perform all the processing.

Our performance is comparable to GLOBAL, and significantly outperforms all the other practical approaches. It incrementally migrates operators and improves the latency by up to 75% for the case of 150 nodes, by placing operators on brokers with smaller network and processing latency. Our protocol applies around 160 migrations, replacing an average of 17 operators at each optimization step.

Constrained metrics

One way of achieving the benefits of pushing all operators closer to the sources, without saturating the processing resources of the root brokers and the nodes close to them, is by adding constraints on the maximum load of the nodes. These constraints will prevent operators from migrating to the root brokers. Moreover, nodes limit their fanout in the tree, as a large fanout would increase the overhead of processing and matching incoming tuples going to downstream nodes. Thus, trees with height higher than one will be constructed. We created such a network by imposing a upper bound of five (5) CPU cycles per time unit on the processing load of each node. Figure 5.12 shows the average output latency of 500 queries on 100 PlanetLab nodes. In this case, not all operators are be migrated to the root since this would violate its upper load limit.

We also used this constrained metric to demonstrate the effectiveness of directed optimization.

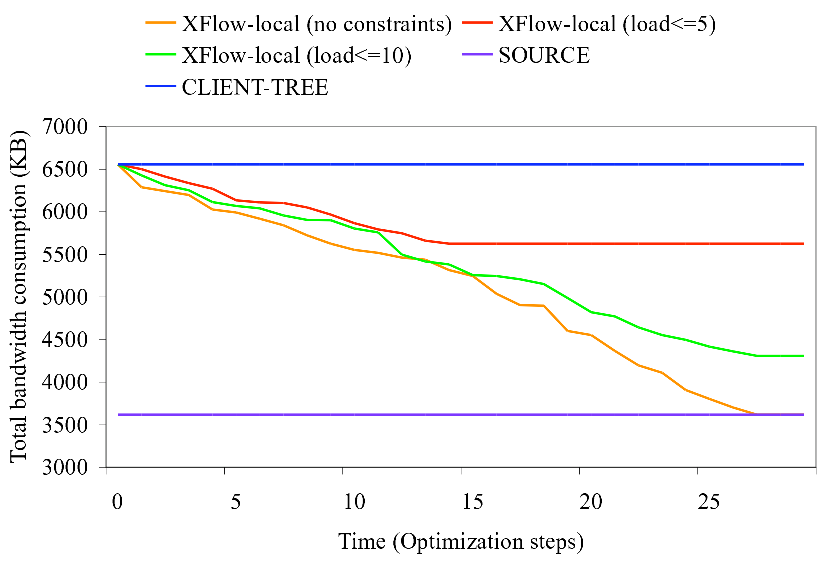


Figure 5.13: Total bandwidth consumption under constraints on the processing load of each node.

Figure 5.12 shows that, when only local migrations are applied, XFlow can improve over its initial placement, provided by CLIENT-TREE, but is outperformed by GLOBAL. This is because our optimization converges to local optimum: nodes cannot migrate their profiles to a better network region because their neighbors cannot accept any new operators without violating their load constraint. However, when directed optimizations are used (*XFlow-hybrid*) nodes do not rely only on their neighbors for gradually improving the system's performance. Network statistics disseminated by their peers informs them about the performance of other nodes, i.e., their latency and load. Hence, XFlow directly migrates operators to nodes with less load and better location. These directed operations allow our system to converge to a configuration that outperforms GLOBAL.

Figure 5.13 shows the total bandwidth consumption when we set the load constraints to 5% and 10% of the node's CPU cycles and when no constraints exist. Our system incrementally improves its performance in all cases. In the absence of constraints, all operators end up being executed at the root brokers. However, when constraints exist, many operators cannot move to higher levels, as this would lead to a violation of the load constraint.

Processing load

We also studied the maximum processing load across all nodes for 500 queries. For this metric, placing all operators on the root brokers (SOURCE) leads to worse performance, as it utilizes all the resources from the root brokers. CLIENT-TREE and CLIENT-STAR distribute the load across all nodes that host a client. However, they perform worse than XFlow and GLOBAL which places every

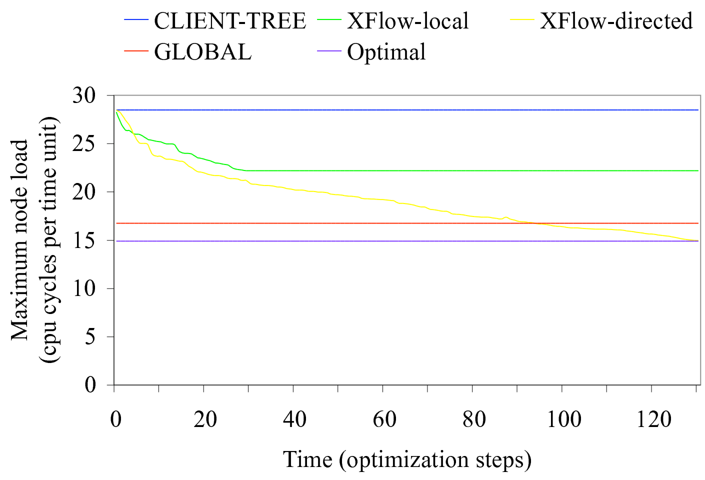


Figure 5.14: Maximum node load with local and directed optimizations.

new query at the least loaded node. For this metric, we also computed the OPTIMAL placement using an exhaustive search of the solution space.

Figure 5.14 shows that XFlow converges to the OPTIMAL when directed optimizations are used over 500 queries and 100 nodes. If only localized optimizations are applied, XFlow can improve its optimization metric but converges to a local optimum. However, by using the network statistics, the most loaded node can easily discover less loaded nodes and migrate part of its local processing, reaching eventually the optimal resource allocation. Moreover, as network size increases (Figure 5.15), XFlow performs even better, since more nodes are available for distributing the processing. In all the cases, we converge to the OPTIMAL and outperform GLOBAL.

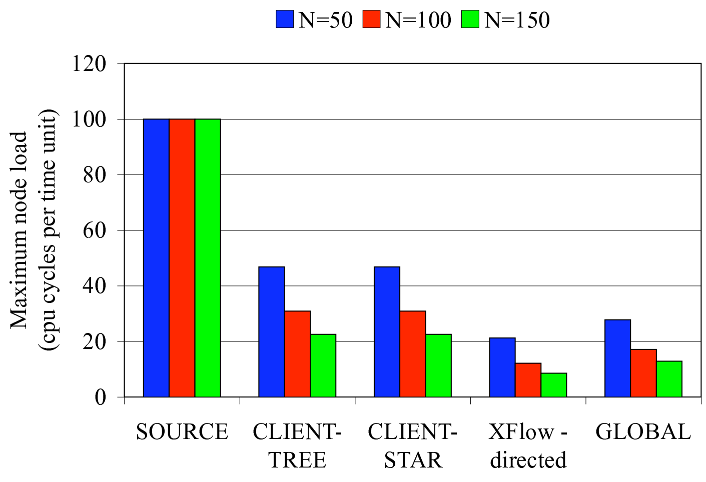


Figure 5.15: Maximum processing load for different network sizes.

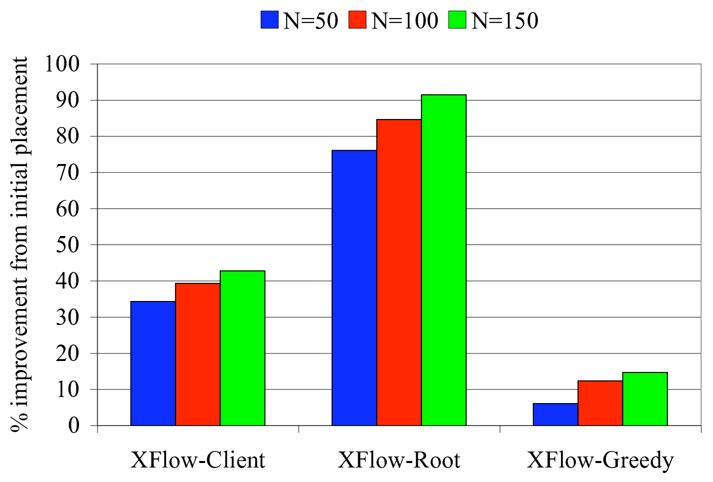


Figure 5.16: Improvement of maximum load for different initial placements.

5.5.2 Effect of initial placement

We use the maximum load metric to study the effect of the initial placement of operators. Figure 5.16 shows the improvement we can achieve compared with the system's performance when we initially place the operators:

- on the clients' hosts (*XFlow-Client*),
- on the root brokers (*XFlow-Root*) and
- on the least loaded node (*XFlow-Greedy*).

XFlow significantly improves the maximum load for all cases. Our protocol migrates as many operators as possible from the root brokers and distributes the processing overhead across the network. XFlow-Client and XFlow-Greedy demonstrate smaller improvements as we already start with a good initial placement. The improvement is higher for the XFlow-Root case since maximum load for the initial performance is already very low (see Figure 5.15). However, even in the case of XFlow-Greedy, we manage to improve the performance by 28% when the network size increases to 150 nodes, by utilizing better the available nodes in the system.

5.5.3 Probabilistic statistics

We also used the maximum load metric to evaluate the effect of our probabilistic statistics dissemination. Figure 5.17 shows the progress of our optimization over 100 nodes when we propagate all statistics (k=100) and the cases where only the top-40 and top-80 values are sent. These values are picked as described in Section 5.4.3. The results show that limited statistics do not incur significant performance degradation as XFlow is able to reach a configuration with performance close to that of OPTIMAL. This fact reveals that the our statistic dissemination approach is able to identify useful statistics for the system's performance. The probabilistic dissemination degrades the performance by at most 10% from the optimal. One the other hand, it manages to decrease the overhead of disseminating these statistics by up to 85% (for network sizes of 200 nodes) (see Figure 5.20). The trade-off between convergence to the optimal and optimization overhead depends on the specific expectations and requirements of the application. However, these results demonstrate that XFlow provides the necessary mechanisms for exploring those two metrics simply by set the k parameter in the system configuration.

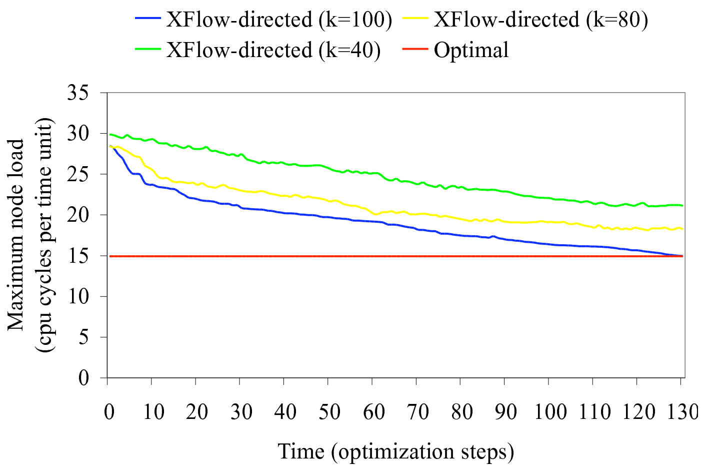


Figure 5.17: Optimization of maximum node load with probabilistic statistics.

5.5.4 Scalability

Due to our aggregation model, XFlow nodes to maintain and exchange state of small size. This allows our system to be highly scalable with the number queries and nodes in our network. We demonstrate XFlow's scalability in Figure 5.18 which shows the traffic incurred for the maximum processing load metric. This traffic includes *optimization traffic*, the *maintenance traffic* (described in Section 5.4.5), and the statistic disseminated to support directed optimizations ((Section 5.4.3). This traffic is a measure of the optimization overhead.

Figure 5.18 shows that XFlow scales with respect to the query set since the network traffic remains constant with the number of queries. Figure 5.19 shows the total network traffic for increasing number of nodes. Larger networks require statistics to be disseminated to more nodes, increasing the traffic with the number of nodes. However, even in this case the maintenance and optimization traffic do not increase and the overhead incurred by statistics dissemination. To reduce this overhead

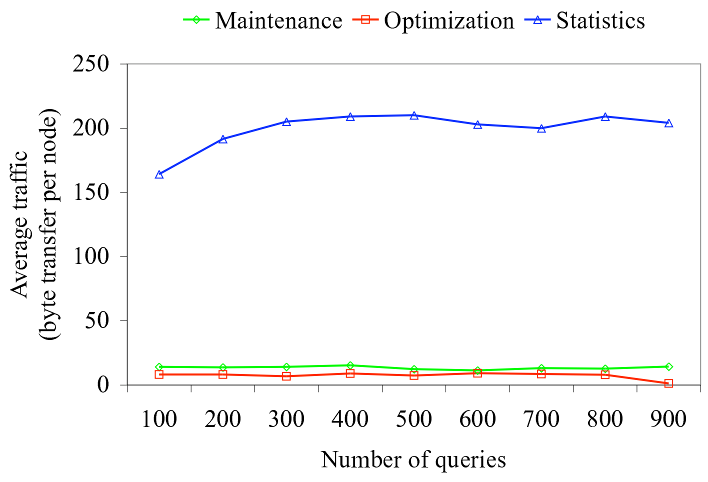


Figure 5.18: Overhead for different number of queries.

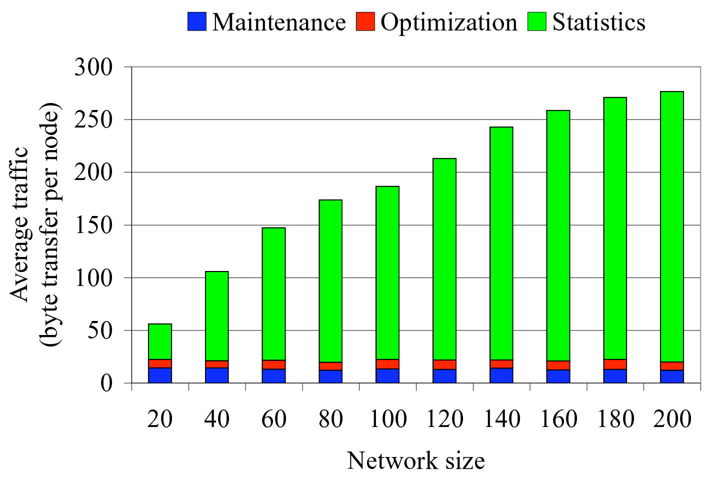


Figure 5.19: Overhead for different network sizes.

we propagate statistics using the probabilistic technique described in Section 5.4.3. Figure 5.20 compares the cases where all statistics all disseminated and when only the top-20 and top-40 are selected. The experiments reveal that the probabilistic dissemination keeps the statistics traffic independent of the number of nodes. Hence, XFlow's overhead is independent of the number of queries and nodes, demonstrating the scalability and efficiency of our approach.

Finally, Figure 5.21 shows the average optimization, maintenance and statistics data sent per node for all the metrics we implemented. The results reveal that XFlow has very small overhead: it requires less than $240 Bytes$ in the worst case, while the optimization traffic remains below $90 Bytes$.

More specifically, for the query latency, each virtual node periodically sends its latency from the root of the tree to its children, in order for them to evaluate their own latency. Similarly for

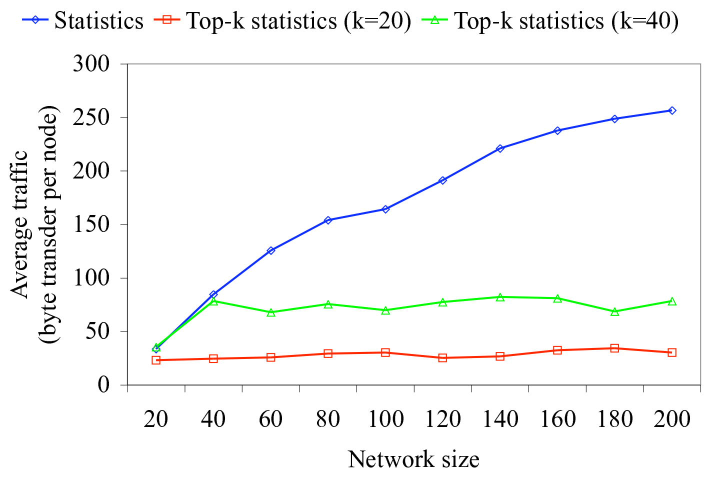


Figure 5.20: Overhead reduction from probabilistic statistics.

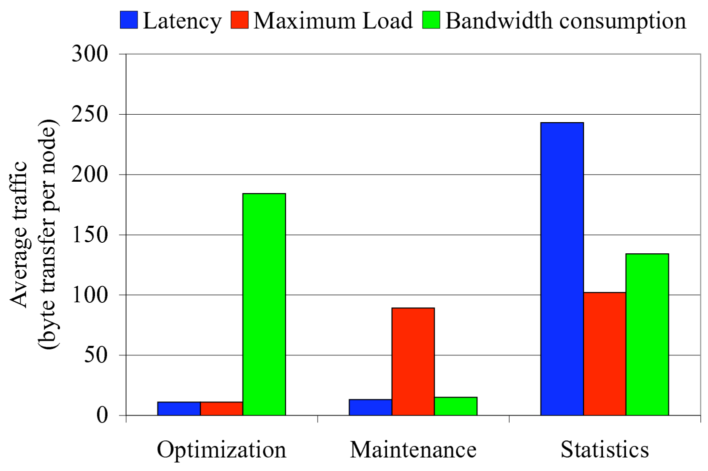


Figure 5.21: Network traffic for different metrics.

bandwidth, virtual nodes send their input rates to their parent in the tree. In the case of maximum load, nodes calculate their processing cost locally. However, data is exchanged in order for nodes to evaluate their local state, e.g., the maximum load of their subtree and the maximum load in the network apart from their subtree.

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

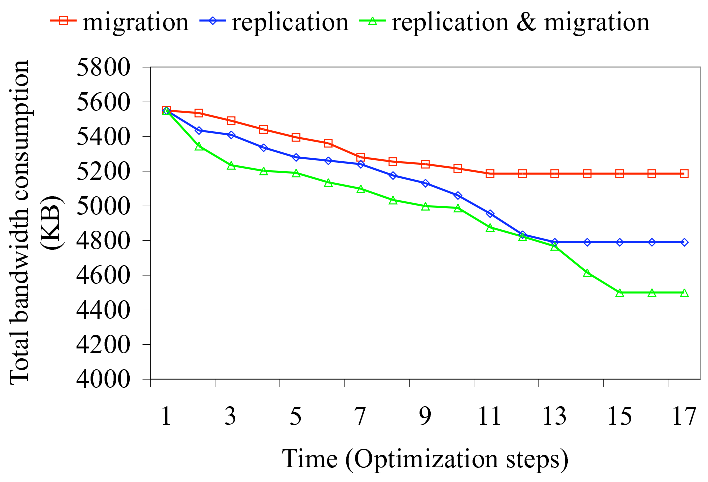


Figure 5.22: Impact of operator replication and migration for the total bandwidth consumption.

5.5.5 Operator replication and partition

We now study our operator execution rules. In this section we examine first replication and then focus on operator partitioning. We allow our operators to be replicated or partitioned only once.

Replication

Figure 5.22 shows the improvement on bandwidth consumption when replication is used for 500 queries. We compare three cases in which we apply: (i) migration, (ii) replication, and (iii) migration and replication together. The improvement achieved with migration is very limited, since moving an operator to another node in one tree might not yield any benefit, as this node may not be subscribed to the second tree. When only replication is used, the performance is better. However, since no migration rules are allowed, these replicas are not reallocated and the bandwidth consumption can not improve further. Not surprisingly, best results are obtained when replication and migration are used together. In this case, XFlow will try to migrate our replicas closer to the sources, if this reduces the data forwarding. Since these replicas take input from a single tree, it is easier to identify a beneficial migration.

Operator partition

Figure 5.23 shows results for the operator partition case when we try to improve the maximum load for 500 queries and 150 nodes. Migrating an operator incurs some benefits, however, since the load of a query depends on its input rate, reducing this rate will reduce its load as well. Partitioning an operator can achieve this, as now half of the input rate is processed by each replica. Moreover,

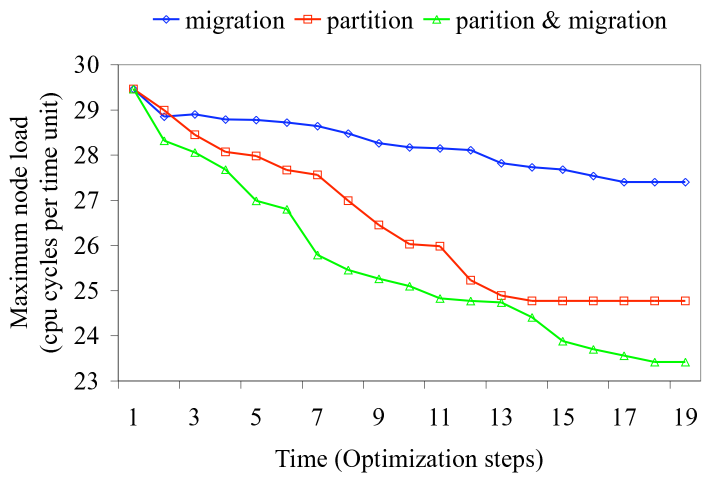


Figure 5.23: Impact of operator partition and migration for the maximum load across nodes.

when migration is allowed, these replicas can move independently in their respective trees, utilizing more processing resources and improving performance.

Chapter 6

Related Work

This chapter outlines the research that is relevant to the design and implementation of an extensible stream dissemination and processing system.

6.1 Extensibility

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 [11] and Starburst [52]. Indeed, the generalization process need not be restricted to the domain of large DBMSs, perhaps best exemplified by GiST [32]. Moreover, recent efforts from the networking community, such as MACEDON [48], Click [36], P2 [41] and IFLOW [38] provide examples of systems promoting the advantages of extensibility. Below we describe the most representative extensible systems in the area of data management and networking.

GiST Hellerstein et al. [32] introduced an index structure, called a Generalized Search Tree (GiST), which is a generalized form of an R-tree. GiST provides a framework generalizing the problem of implementing search indices in a database. The framework includes a template indexing structure that allows domain experts (e.g in computer vision, bioinformatics, or remote sensing) to easily customize a database system to index their content. This package unifies a number of popular search trees in one data structure (i.e., R-trees, B+-trees, hB-trees, TV-trees, Ch-Trees, ranked B+-trees), eliminating the need to build multiple search trees for handling diverse applications. In addition to unifying all these structures, GiST provides both data and query extensibility. To make GiST work, users have to figure out what to represent in the keys and provide the implementation of four methods for the key class that help the tree do insertion, deletion, and search.

In many ways, our work draws its inspiration from GiST, striving to apply the same design principles to distributed data dissemination applications. In XPORT, applications designers are asked to provide the methods implementing the main message matching and dissemination functionality. Our goal is to provide a generalized framework that can be easily customized by domain experts to provide collection, processing and dissemination of their application-specific data types.

MACEDON The goal of MACEDON (Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks) [48] is to facilitate the research and deployment of overlay algorithms. To this end, MACEDON consists of three related pieces: (a) a common overlay network API by which any MACEDON-created overlay implementation may be used, (b) a domain-specific C++ like language to describe an overlay's behavior from which real operating code can be generated and (c) a software engine that implements common features and functionality of overlay algorithms, providing a performance-tuned system that can be employed by any MACEDON-specified overlay. As a result, MACEDON allows algorithm designers to focus their attention on the algorithm itself and less on tedious implementation details. The use of a common, yet extensible, API allows application developers to create a single implementation capable of executing over any overlay algorithm.

CLICK Click [36] is an open, extensible, and configurable router framework. It provides a modular architecture for processing packets in routers using a flow-based configuration specification. Click's building blocks are packet processing modules. To build a router configuration, the user connects a collection of modules into a graph; packets move from element to element along the graphs edges. To extend a configuration, the user can write new modules or compose existing ones in new ways, much as UNIX allows one to build complex applications directly or by composing simpler ones using pipes.

P2 P2 [41] uses a high-level declarative language to express overlay networks in a highly compact and reusable form. Applications submit to P2 a concise logical description of an overlay network, and P2 executes this to maintain routing data structures, perform resource discovery, and optionally provide forwarding for the overlay. P2 is intended to greatly simplify the process of selecting, implementing, deploying and evolving an overlay network design. It is using a declarative logic language for specifying overlays and it employs a dataflow framework at runtime for maintaining overlays instead of the more conventional finite state machine approach. P2 automatically compiles the declarative specification of the overlay into a dataflow program and can compile multiple overlay specifications into a single dataflow.

IFLOW IFLOW [38, 39] is a self-management middleware for developing applications for distributed processing and dissemination of data flows. IFLOW allows applications to express their own performance goals and employs extensible networking monitoring mechanisms to configure and adjust the runtime system behavior. However, their framework expresses utility metrics only as the sum of link costs and they assign stream flows to network edges based on a centralized, exhaustive search of the solution space. Our cost model is more expressive and supports a superset of the optimization metrics considered in IFLOW. Furthermore, we use a decentralized approach that incrementally improves the global cost based on simple, localized views of the network conditions.

Although all the above systems promote the notion of extensibility, to the best of our knowledge, we have yet to see extensible architectures capable of generalizing over the core dissemination and distributed stream processing functionality and optimization objectives. Moreover, our tree-based dissemination architecture allowed us to design a well-defined and structured cost model that relies on tree-related semantics and thus, is able to apply more specific optimizations (network transformations, operator migrations) than any generally structured overlay construction framework.

6.2 Publish-Subscribe systems

Much work has focused on publish subscribe systems including XNet [24, 25], YFilter [29] and ONYX [30] for XML messages and SIENA [20] and GRYPHON [13] for relational data. These systems focus on optimizing specific metrics, e.g., bandwidth usage [25, 20], efficiency of profile matching [13, 20, 29]. Moreover, they commonly support subscriptions with only predicate-based filters over individual events, so they cannot express complex stream-processing queries across multiple messages and streams as XFLOW.

SIENA One of the fist implementations of a distributed content-based publish/subscribe system is the Scalable Internet Event Notification Architecture (SIENA) [18]. SIENA is a multi-broker event notification service that is targeted at Internet-scale deployment. SIENA investigates the publish subscribe model for relational data. An event consists of a set of typed attributes. Subscriptions are conjunctions of event filters, which are predicates over the event attributes. A subscription matches an event if and only if all event filters in the subscription hold when evaluated with respect to the event. SIENA is a promising approach for a large-scale middleware but the topology of the overlay network of event brokers is static and must be specified at deployment time. The efficiency of the content-based routing will therefore depend on the quality of the overlay network topology. Having a static topology is not feasible for a large-scale system, that may involve thousands of event brokers running at geographically dispersed sites.

Gryphon The Gryphon project at IBM Research [13] led to the development of an industrial-strength, reliable, content-based event broker. It is a publish/subscribe middleware implementation that provides a redundant, topic-based and content-based multi-broker publish/subscribe service. Gryphon is based on an information flow model for messaging. An information flow graph (IFG) specifies the exchange of information between information producers and consumers. A logical IFG is mapped onto a physical event broker topology. The topology mapping is statically defined at deployment time, although more recent work includes dynamic topology changes due to failure and evolution. Gryphon's overlay network of event brokers is static and it is defined in configuration files at deployment time. This makes it difficult for the middleware to adapt to changed network conditions.

YFilter YFilter [29] aims to provide fast, on-the-fly matching of XML encoded data to a large number of interest specifications, and transformation of the matching XML data based on client-specific subscriptions. In YFilter, subscriptions are written in a subset of XQuery and are treated as continuous queries, i.e., they will be continuously applied to all incoming messages. These subscriptions, along with the XML messages, are provided as the input to a message broker. The output of the broker is a stream of XML messages to be delivered to users or applications whose interests are satisfied. These XML messages can be the original incoming messages or the customized messages transformed from the incoming ones based on the individual query specifications. To produce the output XML messages, YFilter constructs execution plans for all the queries. When processing a message, the runtime system of YFilter matches all queries with the message, extracts message components for the matched queries, and organizes the extracted components in an intermediate format for efficient translation into customized output messages.

YFilter focuses on improving the matching overhead on each message broker. However, their approach is centralized. All XML messages and XPATH queries are sent to a central location, where the YFilter is deployed and identifies efficiently the final destinations of each incoming message. XPORT allows application designers to define their own structures for storing and indexing subscriptions, thus YFilter's approach can be easily incorporated to allow for efficient message matching and fast forwarding.

ONYX ONYX [30] studies Internet-scale data dissemination that delivers XML-encoded documents from multiple publishing sites to millions of subscribers based on the subscribers' data interests. The system explores the idea of content-based routing of documents in distributed dissemination systems trying to enhance such data dissemination with advanced services such as stateful publish/subscribe and QoS. ONYX employs a network of message brokers that collaboratively provide high scalability and high functionality. Each message broker in this network runs a YFilter [29] instance to filter and transform messages. In addition, each broker contains a routing component to efficiently forward messages to the downstream brokers that are interested in the messages. Interestingly, the routing component is also built on YFilter technology.

While ONYX addresses the problem of distributed XML-based dissemination, it operates over a static overlay network. Brokers are initially connected through a spanning tree, which is not adapted during run time. XPORT's generalized optimization can support dissemination and transformation XML messages, while it can adapt the structure of the network to reflect the most current workload and network conditions.

XNet XNet, an XML-based data dissemination system, uses the XTrie [24] indexing structure for efficiently filtering XML documents based on XPath expressions, and the XRoute [25] protocol for routing XML messages to interested parties. The XTrie index structure offers several features that make it especially attractive for large-scale publish/subscribe systems. It is designed to support effective filtering based on complex XPath expressions, and both ordered and unordered matching of XML data. Finally its matching algorithm is able to reduce the number of unnecessary index probes as well as avoid redundant matchings, thereby providing extremely efficient filtering. It takes advantage of subscription similarities to "aggregate" them in the routing tables, and hence minimize the space requirements and increase the filtering speed at the routers. XPORT can easily exploit the advantages of both XTrie and XRoute by using them for its message indexing and matching respectively.

Our infrastructure provides an abstraction over the matching functionality of publish-subscribe systems and it is able to support both the XPath and relational profiles. In addition, it supports a superset of the optimization metrics considered by the approaches described above. Finally, it allows the distributed execution and continuous optimization of complex stream-based queries by integrating its extensible publish subscribe model with stream processing functionalities.

6.3 Application-level multicast

Several approaches were proposed to build application-level multicast. The general strategy is to construct an information dissemination tree that has the multicast group members as leaf nodes. In contrast with our generalized framework for building application-level dissemination overlays, the approaches described below focus on optimizing specific metrics.

Scribe Scribe [22] is a large-scale event notification infrastructure built on top of Pastry [50]. Scribe uses Pastry to manage multicast groups associated with a topic. Each topic forms a multicast tree depending on the subscribers interested in the topic. Scribe takes advantage of Pastry's locality properties so that the multicast tree is optimized with respect to a proximity metric. Pastry's randomization properties ensure that the multicast tree remains well-balanced. Scribe supports three main operations, which are (1) creating a topic, (2) subscribing to a topic, and (3) publishing a message on a topic. The lack of content-based filtering of publications in Scribe limits the expressiveness of subscriptions and makes Scribe unsuitable as a general-purpose publish-subscribe middleware.

SplitStream SplitStream [21] is a high-bandwidth content distribution system based on end-system multicast. It distributes the forwarding load among all the participants, has increased robustness to node failures, and is able to accommodate participating nodes with different bandwidth capacities. The key idea in SplitStream is to split the content into k stripes and to multicast each stripe using a separate tree. Peers join as many trees as the stripes they wish to receive and they specify an upper bound on the number of stripes that they are willing to forward. The challenge is to construct this forest of multicast trees such that an interior node in one tree is a leaf node in all the remaining trees and the bandwidth constraints specified by the nodes are satisfied. This ensures that the forwarding load can be spread across all participating peers.

Bullet Bullet [37] is a system for high bandwidth data dissemination (i.e., multimedia streaming and content distribution) for large-scale distributed systems. It uses a scalable and distributed algorithm that enables nodes spread across the Internet to self-organize into a high bandwidth overlay mesh consisting of multiple overlay trees. Each Bullet node transmits a disjoint set of data to each of its children, with the goal of maintaining uniform representativeness of each data item across all participants. The level of disjointness is determined by the bandwidth available to each of its children. Bullet then employs a scalable and efficient algorithm to enable nodes to quickly locate multiple peers capable of transmitting missing data items to the node. Their approach increases the system's throughput by enabling clients to receive different data segments from multiple parents in the mesh. Our system model for supporting stream processing applications relies on the similar key idea: nodes are organized in an overlay mesh consisting of multiple overlay dissemination trees. Each node receives data from multiples tree increasing the throughput and the robustness of the system.

Bayes An application-level multicast solution is Bayeux [57]. Bayeux is based on the Tapestry [54]. Its algorithm is similar to Scribe's in the sense that it uses an overlay routing layer to build a tree for a multicast topic. However, in Bayeux, a subscription message is routed all the way to the root of the multicast tree. The root node updates its membership list for the topic and replies with a tree message destined for the new subscriber. The tree message then creates state at the forwarding nodes along its path and ensures that the new subscriber becomes part of the multicast tree. Bayeux's approach is less scalable because the root node has to keep membership information about all nodes belonging to a multicast group. In addition, group membership management is more costly as subscription messages are routed to the root node and trigger tree messages. To alleviate the problem of the root node becoming a bottleneck, a scheme for partitioning the multicast tree is suggested so that the load can be shared among several roots.

Multicast over CAN A multicast mechanism over CAN is presented in [47]. The strategy adopted is different from the previous approaches since no tree is constructed in the overlay network. Instead, a separate content-addressable network within the global CAN is formed that only contains the multicast group members. To publish a message to the group, the multicast CAN is flooded with the publication so that all group members receive it. The flooding algorithm takes advantage of the structure of the n-dimensional coordinate space of CAN node IDs to minimize the delivery of duplicate messages. However, a comparison has shown that this scheme is less efficient than tree-based multicast approaches [23].

Overcast Several application-level multicast schemes, such as Overcast [35], exist that do not rely on a distributed hash table. Instead, a scalable and reliable single-source multicast service is provided by directly building a multicast tree out of Overcast nodes that are distributed in the network. A new node joins the multicast tree by first contacting the root node. It recursively obtains a list of children and then computes a proximity metric in order to find an optimal location for itself in the tree. This adds a higher overhead to joining a multicast group compared to DHT-based multicast. Moreover, Overcast optimization goal is to maximize the bandwidth availability which can be easily expressed through our framework's grammar.

6.4 Distributed stream processing

Our framework is directly relevant to distributed stream processing. In general, these systems are designed and evaluated in small-scale cluster environments, thus, they do not address scalable and adaptive data collection and distribution. They also lack extensibility in terms of the optimization metrics they support. Finally, none of these systems allows for operators to be replicated or partitioned. Below we described the most representative approaches.

SBON A Stream-Based Overlay Network (SBON) [45] is an infrastructure that manages and optimizes stream queries from multiple applications. SBON performs an operator placement decision, creating a mapping of operators to physical overlay nodes. This mapping should make efficient use of network resources, for example, by filtering data close to the sources. The SBON uses a decentralized algorithm for network-aware operator placement called Relaxation placement, which finds a solution in two steps. First, an unpinned operator in a query is placed using a spring relaxation technique in a virtual metric latency space. After that, the solution is mapped to actual physical overlay nodes. The function minimized by this approach is the data rate-latency product. This product is the amount of data in transit in the network and thus a measure for global network usage.

SAND SAND [9], proposes a set of approaches or in-network placement of stream processing operators. Operators are placed either at the consumer side, at the producer side, or in-network on a DHT routing path between the two endpoints, depending on the bandwidth usage of a query. Applications can also specify delay constraints on the placement path in the DHT. Our approach of performing operator placement is more general than SAND because placements are not tied to DHT routing paths or to a specific optimization metric. Moreover, previous work [46] has shown that DHT routing paths can lead to inefficient candidate sets for operator placement. This is because DHT routing tables are optimized for minimizing hop count and not for delay or bandwidth usage.

Borealis Borealis [6] is a distributed stream-based processing system that inherits core stream processing functionality from Aurora [7] and distribution capabilities from Medusa [12]. Borealis includes a optimization framework that includes three levels of collaborating optimizers. At the lowest level, a local optimizer runs at every site and is responsible for scheduling messages to be processed as well as deciding where in the locally running diagram to shed load, if required. A neighborhood optimizer also runs at every site and is primarily responsible for load balancing the resources at a site with those of its immediate neighbors. At the highest level, a global optimizer is responsible for accepting information from the end-point monitors and making global optimization decisions. Although Borealis supports run-time operator migration (but not operator replication or partitioning), it currently avoids the operator placement problem by supporting only pre-defined operator locations with pinned operators in the network. This leaves the burden of efficient operator placement to the system administrator, which is infeasible for a dynamic, large-scale system with thousands of queries.

Medusa Medusa [12] is a distributed mechanism for managing load in a federated system. Its mechanism is based on private pairwise contracts negotiated offline between participants. Contracts set tightly bounded prices for migrating each unit of load between two participants and specify the set of tasks that each is willing to execute on behalf of the other. Medusa provides a simple and lightweight runtime load management using price pre-negotiation while maintaining good system-wide load balance properties. The load transfer mechanism is simple: a participant moves load to another if the local processing cost is larger than the payment it would have to make to another participant for processing the same load. Although their load management mechanism is motivated by federated distributed stream processing, it also applies to other federated systems such as Web services, computational grids, overlay-based computing platforms, and peer-to-peer systems. However, Medusa's approach while appropriate within a single data center, it may leads to poor performance on a wide-area network, in which communication latencies can dominate processing costs.

PIER PIER (Peer-to-peer Information Exchange and Retrieval) is a distributed query engine based on overlay networks. It employs a generic data flow engine which supports a set of relational query processing operators. It includes multi-hop, in-network algorithms for join, aggregation, and query/result dissemination. PIER can support both traditional query trees and DAGs, as well as cyclic graphs representing recursive queries. In addition to a "boxes and arrows" data-flow interface, PIER also provides a simple SQL interface and support for continuous query variants of SQL. PIER aims to build an Internet-scale querying system on top of a routing network given by a DHT, where operators are placed randomly. Although random distribution of operators has good load-balancing properties, it may cause large query delays when operators are placed at nodes distant from both producers and consumers.

6.5 Continuous adaptive optimization

Closely related to our work are approaches that use the concept of local transformations to perform continuous adaptive optimizations of the dissemination tree [14, 49, 55]. Below we describe the most representative of these.

Zhou et al. [55] construct a dissemination tree that optimizes *fidelity*, a metric that captures the dissemination efficiency in terms of the satisfaction of coherence requirements, taking into account both the communication and processing cost incurred during dissemination. The performance of the dissemination tree is continuously monitored and its structure adapts to changing conditions through a set of localized tree transformations. In this work the localized transformations are designed for specific metric of the system, which can be expressed in our generalized framework.

OMNI OMNI [14] (Overlay Multicast Network Infrastructure) consists of a set of nodes organized into an overlay providing the multicast data delivery backbone that distributes data to a set of end-hosts. It employs a distributed iterative scheme that constructs good data distribution paths on the OMNI. Their scheme allows a multicast service provider to deploy a large number of nodes without explicit concern about optimal placement. Once the capacity constraints of the nodes are specified, OMNI organizes them into an overlay topology, which is continuously adapted with changes in the distribution of the clients as well as changes in network conditions. OMNI optimizes the end-to-end latency delivered on the multicast trees, and thus, their iterative network transformations are designed for a specific metric, as opposed to the general optimization framework of XPORT. Moreover, their metric can be easily expressed through the grammar and aggregation model of XPORT.

AMMO AMMO [49] (Adaptive Multi Metric Overlays) provides a framework for constructing adaptive multi-metric overlay networks. An application specifies its performance criteria using a metric function, performance constraints and constraint priorities. The metric function assigns a metric weight to an edge and is used to guide overlay transformations once the overlay has does not satisfy the application constraints. 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.

Chapter 7

Conclusions

This dissertation introduces general-purpose, extensible solutions designed to support a wide range of stream processing and dissemination applications. It is motivated by the observation that these applications often exhibit diverse logic and performance requirements, yet they all require common facilities, which include construction of an overlay network, routing and processing logic, and membership management. In contrast to existing approaches that provide custom, point solutions to point applications, we introduce a generic infrastructure that provides these core functionalities and can be easily extended for a broad spectrum of target applications.

Our basic infrastructure can be customized to support diverse profile management logic, stream types, and performance targets. This is achieved through a set of methods that encapsulate application-specific data types and behavior and a declarative cost model for defining the desired performance metrics. We proposed a novel, generic, metric definition model that relies on the aggregation of cost metrics defined over the system entities (e.g., nodes, overlay links/paths). This model allows for the uniform specification of many commonly used performance measures, as well as new ones, through combinations of different aggregation functions and metrics.

Based on the desired performance goals, the system progressively refines its performance using metric-independent decentralized algorithms. A key feature of our approach is that it relies on localized state and interactions to reduce the global system cost. Our framework utilizes the metric definition model that allows us to rely only on localized, aggregated, network and workload statistics. Nodes apply our optimization algorithms on their neighborhood as well as on specific promising network regions which they discover through dissemination of these statistics. One of our key results is that even simple aggregations of statistics are sufficient to achieve efficient operation with low overhead, as well as allow our approach to avoid local optimum and converge to near-optimal configurations.

Based on our generic infrastructure, we designed two extensible systems which achieve a clean separation between the “plumbing” and “application” and enable the uniform support of disparate dissemination and monitoring applications. In the next paragraphs, we summarize the conclusions for each of these systems.

7.1 XPORT: Extensible Profile-driven Stream Dissemination

Profile-driven dissemination applications match the content of generated streams against the clients subscriptions and identify the network routes connecting the data sources to their subscribers. Example applications include multi- player online games, multicast-based content distribution and web feed dissemination. To support these applications, we designed and developed XPORT, a generic publish-subscribe distributed stream dissemination system. The system consists of an overlay-based dissemination tree, which connects data sources with the interested parties and provides content-based stream routing functionality.

In the context of XPORT, we studied network optimizations, which refine the structure of the dissemination tree by modifying the overlay connections among the system nodes. This set of optimizations includes pre-defined primitive network transformations as well as application-defined composite transformations obtained through the composition of the primitive ones. We apply these optimizations on each node’s localized neighborhood as well as across non-local network areas that demonstrate good performance. Using XPORT, we built a peer-to-peer RSS feed dissemination service and a multi-player networked game, both of which can be optimized for a variety of metrics including total path latency, variance of path latency, average bandwidth consumption, bandwidth bottleneck, and total received redundant data. We evaluated XPORT by deploying its prototype on the PlanetLab testbed. These experiments allowed us to verify our systems flexibility and effectiveness, as it managed to improve each of these metrics significantly. XPORT was able to converge to network topologies that perform very close to the optimal configuration, while incurring low optimization overhead.

7.2 XFlow: Extensible Stream Processing

To support a broad spectrum of Internet-scale monitoring applications, we designed XFlow, an extensible, highly-scalable and adaptive framework for distributing and optimizing stream processing queries. XFlow leverages our generic metric definition model in order to express measures that evaluate the efficiency of user queries and the utilization of the system resources. It creates and optimizes an overlay network of multiple overlay dissemination trees, given dynamic stream sources, clients with stream-oriented complex queries and application-defined QoS and resource-utilization expectations and constraints.

XFlow progressively refines the placement and execution of the operators, the structure of the underlying overlay network, as well as the statistics collection process, to meet the desired objectives. This generic optimization framework relies on localized state and interactions in order to reduce the global system cost. The optimizations are guided by a set of operator distribution operations (migration, replication, partition) and use aggregated, network and workload statistics. Nodes apply these operations on their neighborhood, as well as on specific network regions which they discover through dissemination of statistics. Moreover, we employ probabilistic techniques for disseminating statistics, keeping the network traffic within constant bounds independently of the number of queries and nodes and allowing for high scalability and efficiency. We developed an initial prototype of XFlow and evaluated its performance on the PlanetLab testbed. Our experiments showed that XFLOW can significantly improve various QoS and resource utilization metrics with low overhead and it managed to consistently avoid local optimum and converge to near-optimal configurations.

7.3 Future directions

This thesis proposes abstractions that support application-specific performance targets and profile-management logic. While these solutions facilitate the design of extensible distributed stream dissemination and processing systems, there exist many more applications that could benefit from a clean separation of application logic and data management. Emerging applications, such as feed-based mash-ups and scientific collaborative systems, require domain-specific processing logic. These demands are currently satisfied through user-defined functions. One immediate research direction is to support the distributed implementation of such user-defined processing operators. This extension introduces new challenges since operator distribution techniques, such as replication and partition, are tightly coupled with the operators semantics. One approach would be to define semantic-agnostic optimizations by providing a hinting interface through which users can specify relevant properties of their operators, for example, how their functions can be parallelized.

Our infrastructure employs optimization protocols that deliver exact query results. However, producing approximate query answers to improve performance can provide more flexible systems and services. For example, systems with limited bandwidth, memory or processing resources could benefit from strategic in-network placements of load-shedding and projection operators which discard stream tuples or unnecessary tuple attributes, respectively. Here, the main challenge is to design a generic utility-aware cost model that expresses the trade-offs between the desired degree of the approximation of results and the performance expectations of the application.

An interesting future direction is to combine the two different types of optimization rules discussed in this thesis, i.e., network and query optimizations. Both XPORT and XFlow operate over the same goal that is to improve the system cost. However, they follow different approaches to achieve that. XPORT changes the structure of the dissemination tree, while XFlow changes the deployment and implementation of the query operators. It would be interesting to explore the benefits and trade-offs of combining these approaches. The main challenge here is the amount of state required to support both types of optimizations. The straightforward approach is to maintain the state required by each type of optimization rules independently. However, we believe the state of both optimizations can be more efficiently combined. Both approaches rely on the same fundamental idea: we identify the system components affected by the optimization and maintain information about their cost metrics. Whenever possible, we aggregate this information to reduce the optimization state. It would be interesting to abstract over these commonalities and identify the state required for both optimization types. This will allow a system to gain the benefits of both systems, while maintaining a low optimization traffic and maintenance cost.

Bibliography

[1] Akamai, http://www.akamai.com/.

[2] Distributed intrusion detection, http://www.dshield.org.

[3] Distributed monitoring framework, http://dsd.lbl.gov/dmf.

[4] Earth scope, http://www.earthscope.org.

[5] Yahoo pipes, http://pipes.yahoo.com/pipes/.

[6] Daniel Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The design of the Borealis stream processing engine. In *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR)*, 2005.

[7] Daniel Abadi, Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A new model and architecture for data stream management. In *VLDB journal*, 2003.

[8] Micah Adler, Zihui Ge, James F. Kurose, Don Towsley, and Stephen Zabele. Channelization problem in large scale data dissemination. In *Proceedings of the 9th International Conference of Network Protocols (ICNP)*, 2001.

[9] Yanif Ahmad and Ugur Cetintemel. Network-aware query processing for stream-based applications. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*, 2004.

[10] David G. Andersen, Hari Balakrishnan, M. Frans Kaashoek, and Robert Morris. Resilient overlay networks. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[11] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim Gray, Patricia P. Griffiths, W. Frank King III, Raymond A. Lorie, Paul R. McJones, James W. Mehl, Gianfranco R. Putzolu, Irving L. Traiger, Bradford W. Wade, and Vera Watson. System R: Relational approach to database management. *ACM Transactions on Database Systems*, 1(2):97–137, 1976.

[12] Magdalena Balazinska, Hari Balakrishnan, and Michael Stonebraker. Contract-based load management in federated distributed systems. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data(SIGMOD)*, 2005.

[13] Guruduth Banavar, Tushar Chandra, Bodhi Mukherjee, and Jay Nagarajarao. An efficient multicast protocol for content-based publish-subscribed systems. In *Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS)*, 1999.

[14] Suman Banerjee, Christopher Kommareddy, Koushik Kar, Samrat Bhattacharjee, and Samir Khuller. Construction of an efficient overlay multicast infrastructure for real-time applications. In *Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, 2003.

[15] Jason Campbell, Phillip B Gibbons, and Suman Nath. IrisNet: an internet-scale architecture for multimedia sensors. In *Proceedings of ACM Multimedia*, 2005.

[16] Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring Streams A New Class of Data Management Applications. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, 2002.

[17] Antonio Carzaniga, David S. Rosenblum, and Alexander L.Wolf. Challenges for distributed event services: Scalability vs. expressiveness. In *In Proceedings of Engineering Distributed Objects (EDO)*, 1999.

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

[19] Antonio Carzaniga, Matthew J. Rutherford, and Alexander L. Wolf. A routing scheme for content-based networking. In *Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, Hong Kong, China, 2004.

[20] Antonio Carzaniga and Alexander L. Wolf. Forwarding in a content-based network. In *Proceedings of the 2003 ACM SIGCOMM Conference*, pages 163–174, 2003.

[21] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony I. T. Rowstron, and Atul Singh. Splitstream: High-bandwidth multicast in cooperative environments. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 298–313, 2003.

[22] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communication (JSAC)*, 20(8), October 2002.

[23] Miguel Castro, Michael B. Jones, Anne-Marie Kermarrec, Antony Rowstron, Marvin Theimer, Helen Wang, and Alec Wolman. An evaluation of scalable application-level multicast built using peer-to-peer overlays. In *In Proceedings of INFOCOM*, 2003.

[24] Chee Yong Chan, Pascal Felber, Minos N. Garofalakis, and Rajeev Rastogi. Efficient filtering of XML documents with xpath expressions. *VLDB Journal*, 11, 2002.

[25] Raphaël Chand and Pascal Felber. Scalable protocol for content-based routing in overlay networks. In *Proceedings of the 2nd IEEE International Symposium on Network Computing and Applications (NCA)*, 2003.

[26] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael Franklin, Joseph Hellerstein, Wei Hong, Sailesh Krishnamurthy, Sam Madden, Vijayshankar Raman, Fred Reiss, and Mehul Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR))*, 2003.

[27] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data(SIGMOD)*, 2000.

[28] Yogen K. Dalal and Robert M. Metcalfe. Reverse path forwarding of broadcast packets. *Communications of the ACM*, 21(12):1040–1047, 1978.

[29] Yanlei Diao and Michael J. Franklin. Query processing for high-volume XML message brokering. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB)*, pages 261–272, 2003.

[30] Yanlei Diao, Shariq Rizvi, and Michael J. Franklin. Towards an Internet-scale XML dissemination service. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*, pages 612–623, 2004.

[31] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[32] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. In *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB)*, pages 562–573, 1995.

[33] Yang hua Chu, Sanjay G. Rao, Srinivasan Seshan, and Hui Zhang. A Case for End System Multicast. In *Proceedings of ACM Sigmetrics*, pages 1–12, 2000.

[34] Navendu Jain, Lisa Amini, Henrique Andrade, Richard King, Yoonho Park, Philipe Selo, and Chitra Venkatramani. Design, implementation and evaluation of the linear road benchmark of the stream processing core. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data(SIGMOD)*, 2006.

[35] John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and James W. O'Toole, Jr. Overcast: Reliable multicasting with an overlay network. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, 2000.

[36] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[37] Dejan Kostic, Adolfo Rodriguez, Jeannie R. Albrecht, and Amin Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 282–297, 2003.

[38] Vibhore Kumar, Zhongtang Cai, Brian F. Cooper, Greg Eisenhauer, Karsten Schwan, Mohamed Mansour, Balasubramanian Seshasayee, and Patrick Widener. Implementing diverse messaging models with self-managing properties using iflow. In *3rd IEEE International Conference on Autonomic Computing (ICAC)*, 2006.

[39] Vibhore Kumar, Brian F. Cooper, Zhongtang Cai, Greg Eisenhauer, and Karsten Schwan. Resource-aware distributed stream management using dynamic overlays. In *ICDCS*, 2005.

[40] Richard Kuntschke, Bernhard Stegmaier, Alfons Kemper, and Anglelika Reiser. StreamGlobe: Processing and sharing data streams in grib-based P2P infrastructures. In *Proceedings of the 31th International Conference on Very Large Data Bases (VLDB)*, 2005.

[41] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing declarative overlays. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 75–90, 2005.

[42] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, approximation, and resource management in a data stream management system. In *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR)*, 2003.

[43] Olga Papaemmanouil, Yanif Ahmad, Ugur Cetintemel, John Jannotti, and Yenel Yildirim. XPORT: Extensible profile-driven overlay routing trees (demonstration). In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data(SIGMOD)*, 2006.

[44] Olga Papaemmanouil and Ugur Çetintemel. Semcast: Semantic multicast for content-based data dissemination. In *Proceedings of the 21st International Conference on Data Engineering (ICDE)*, pages 242–253, 2005.

[45] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-Aware Operator Placement for Stream-Processing Systems. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*, 2006.

[46] Peter Pietzuch, Jeff Shneidman, Jonathan Ledlie, Matt Welsh, Margo Seltzer, and Mema Roussopoulos. Evaluating DHT-Based Service Placement for Stream-Based Overlays. In *Proceedings of the 4th International Workshop on Peer-to-Peer Systems (IPTPS)*, 2005.

[47] Sylvia Ratnasamy, Mark Handley an d Richard Karp, and Scott Shenker. Application-Level Multicast Using Content-Addressable Networks. In *Proceedings of 3rd International Workshop on Networked Group Communication (NGC)*, 2001.

[48] Adolfo Rodriguez, Charles Killian, Sooraj Bhat, Dejan Kostic, and Amin Vahdat. MACEDON: Methodology for automatically creating, evaluating, and designing overlay networks. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI)*, pages 267–280, 2004.

[49] Adolfo Rodriguez, Dejan Kostic, and Amin Vahdat. Scalability in adaptive multi-metric overlays. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS)*, pages 112–121, 2004.

[50] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[51] Daniel Sandler, Alan Mislove, Ansley Post, and Peter Druschel. Feedtree: Sharing web micronews with peer-to-peer event notification. In *Proceedings of the 4th International Workshop on Peer-to-Peer Systems (IPTPS)*, Ithaca, New York, February 2005.

[52] Peter M. Schwarz, W. Chang, Johann Christoph Freytag, Guy M. Lohman, John McPherson, C. Mohan, and Hamid Pirahesh. Extensibility in the starburst database system. In *Proceedings of the International Workshop on Object-Oriented Database Systems (OODBS)*, pages 85–92, 1986.

[53] Carl A. Waldspurger and William E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *OSDI*, 1994.

[54] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A Resilient Global-Scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 22(11), 2004.

[55] Yongluan Zhou, Beng Chi Ooi, Kian-Lee Tan, and Feng Yu. Adaptive reorganization of coherency-preserving dissemination tree for streaming data. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*, 2006.

[56] Yali Zhu, Elke R. Rundensteiner, and George T. Heineman. Dynamic plan migration for continuous queries over data streams. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data(SIGMOD)*, 2004.

[57] Shelley Q. Zhuang, Ben Y. Zao, Anthony D. Joseph, Randy H. Katz, and John D. Kubiatowicz. Bayeux: An Architecture for Scalable and Fault-tolerant Wide-area Data Dissemination. In *Proceedings of the 11th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, 2001.