Towards an Internet-Scale XML Dissemination Service

Yanlei Diao, Shariq Rizvi, Michael J. Franklin

University of California, Berkeley
{diaoyl, rizvi, franklin}@cs.berkeley.edu

Abstract

Publish/subscribe systems have demonstrated the ability to scale to large numbers of users and high data rates when providing content-based data dissemination services on the Internet. However, their services are limited by the data semantics and query expressiveness that they support. On the other hand, the recent work on selective dissemination of XML data has made significant progress in moving from XML filtering to the richer functionality of transformation for result customization, but in general has ignored the challenges of deploying such XML-based services on an Internet-scale. In this paper, we address these challenges in the context of incorporating the rich functionality of XML data dissemination in a highly scalable system. We present the architectural design of ONYX, a system based on an overlay network. We identify the salient technical challenges in supporting XML filtering and transformation in this environment and propose techniques for solving them.

1 Introduction

A large number of emerging applications, such as mobile services, stock tickers, sports tickers, personalized newspaper generation, network monitoring, traffic monitoring, and electronic auctions, has fuelled an increasing interest in *Content-Based Data Dissemination* (CBDD). CBDD is a service that delivers information to users (equivalently, applications or organizations) based on the correspondence between the content of the information and the user data interests. Figure 1 shows the context in which a data dissemination system providing this service operates. Users subscribe to the service by providing *profiles* expressing their data interests. Data sources publish their data by pushing messages to the system. The system delivers to each user the messages that match her data interests; these messages are presented in the format required by the user.

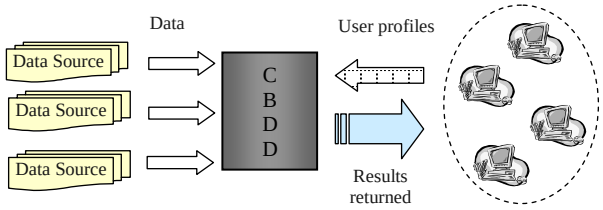

Fig. 1. Overview of content-based data dissemination

Over the past few years, XML has rapidly gained popularity as the standard for data exchange in enterprise intranets and on the Internet. The ability to augment data with semantic and structural information using XML-based encoding raises the potential for more accurate and useful delivery of data. In the context of XML-based data dissemination, user profiles can involve constraints over both the structure and value of XML fragments, resulting in potentially more precise filtering of XML messages. In many emerging applications, the relevant XML messages also need to be transformed for data and application integration, personalization, and adaptation to wireless devices.

While XML filtering and transformation has aroused significant interest in the database community [2][8][12][16][20][22][26], little attention has been paid to deploying such XML-based dissemination services on an Internet-scale. In the latter scenario, services are faced with high data rates, large profile population, variable query life span, and tremendous result volume. Distributed publish/subscribe systems developed in the networking community [1][4][9][10][29] have demonstrated their scalability in applications such as sports tickers at the Olympics [21]. Integrating XML processing into such distributed environments appears to be a natural approach to supporting large-scale XML dissemination.

1.1 Challenges

Distributed pub/sub systems partition the profile population to multiple nodes and direct the message flow to the nodes hosting profiles based on the content of messages (referred to as *content-driven routing*). Integrating XML into content-driven routing, however, brings the following key challenges.

- As XML mixes structural and value-based information, content-driven routing needs to support constraints over both. The inherent repetition and recursion of element names in XML data also defeats well-known routing

This work was funded in part by the NSF under ITR grants IIS-0086057 and SI-0122599, by the IBM Faculty Partnership Award program, and by research funds from Intel, Microsoft, and the UC MICRO program.

Proceedings of the 30th VLDB Conference,
Toronto, Canada, 2004

techniques (e.g., the counting algorithms [10][19]) designed for simpler data models. New techniques for XML-based content-driven routing are needed.
- When XML transformation is introduced to a distributed system, the best venue to perform such transformation is another issue to address.
- The criteria used to partition user profiles have an impact on the effectiveness of content-driven routing. The mixture of structure and value-based constraints in profiles and the repetition of element names in XML data complicate the profile partitioning problem.
- As the verbosity of XML results in large messages and these large messages need to be parsed at each routing step, alternative formats should be considered for efficient XML transmission.

A number of XML query processors are available for providing XML processing in this environment. Among them, YFilter [16][17], a multi-query processor that we built previously, represents a set of profiles using an operator network on top of a *Non-Deterministic Finite Automaton* (NFA) to share processing among those profiles. Using YFilter for distributed XML dissemination raises the issues of distributing the NFA-based operator network, and efficient scheduling of the operators for both profile processing and content-driven routing.

1.2 Contributions

In this paper, we present the initial design of ONYX (*Operator Network using YFilter for XML dissemination*), a large-scale dissemination system that delivers XML messages based on user specifications for filtering and transformation. The contributions of our work include the following.

- We leverage the YFilter processor for content-driven routing. In particular, we use the NFA-based operator network to represent routing tables, and provide an initial solution to constructing the routing tables from the distributed profile population.
- We address the issue of how to perform incremental message transformation in the course of routing.
- In order to boost the effectiveness of routing, we provide an algorithm that partitions the profile population based on exclusiveness of data interests.
- We develop holistic message processing for sharing the work among various processing tasks at a node (i.e., content-driven routing, incremental transformation and user profile processing). Dependency-aware priority scheduling is used to support such sharing while providing a fast path for routing.
- We investigate various formats for efficient XML transmission.
- Last but not least, we provide an architectural design of the system and mechanisms for building such a system.

The paper proceeds as follows. Section 2 details the requirements and motivation. Section 3 describes our system model. Core techniques addressing the various challenges are presented in Section 4, followed by a detailed broker architecture design in Section 5. Section 6 includes extended related work. Section 7 concludes the paper.

2 Requirements and Motivation

In this section, we present the requirements for large-scale XML dissemination, and provide a brief survey of existing solutions, which motivates our work presented in this paper.

2.1 Expressiveness

A starting point for our requirements is the use of XML as the data model and a subset of XQuery [7] as the profile model. User profiles can contain constraints over both structure (using path expressions) and value (using value-based predicates) of XML fragments. For example, if a user is interested in stock information distributed in San Francisco and under the subject "Stock", she can express her interest using the query below (based on the NITF DTD [23]). It specifies that the root element *nitf* must (1) have a child element *head* that in turn contains a child element *pubdata* whose attribute *edition.area* has the value "SF", and (2) have a descendant element *tobject.subject* whose attribute *tobject.subject.type* has the value "Stock".

```
$msg/nitf [head/pubdata[@edition.area = "SF"]]
          [.//tobject.subject[@tobject.subject.type = "Stock"]]
```

User profiles can also contain specifications for result customization. For example, a user can use the query below to specify that for each NITF article that matches the *for* and *where* clauses (which are equivalent to the query above), transform it to a new article with the root element *stock_news* containing elements selected from the original article using path expressions "*body/body.head/hedline*", and "*body/body.content*".

```
for     $n  in   $msg/nitf
where   $n/head/pubdata/@edition.area = "SF"
  and   $n//tobject.subject/@tobject.subject.type = "Stock"
return  <stock_news>
                {$n/body/body.head/hedline}
                {$n/body/body.content}
        </stock_news>
```

As the profile model is based on the XQuery language, in the sequel, we use the terms profile and query interchangeably.

2.2 Scalability

The second dimension of requirements is scalability. More specifically, the service must scale along the following dimensions.

Data volume. The data volume is determined by the number of messages per second arriving at the system and the message size. Depending on the application, the number of messages per second ranges from several to thousands. For example, *NASDAQ* real-time data feeds include 3,000 to 6,000 messages per second in the pre-market hours [43]; Network and application monitoring systems such as *NetLogger* can also receive up to a thousand messages per second [44]. The message size can vary from 1 KB (e.g., XML encoded stock quote updates) to 20 KB (e.g., XML news articles).

Query population. The query population in a dissemination system can also span a wide range, reaching millions of

queries for applications such as personalized newspaper generation and mobile operators providing stock quote updates.

Frequency of query updates. A third scalability issue is the frequency with which users update their data interests. While in some applications queries change on a daily basis, in some others they can change much more frequently.

Result Volume. When result customization is supported, the volume of results to be delivered can be tremendous. This is because for each message, point-to-point delivery is needed for every query matched by the message. Take, for example, a stock quote update service. Suppose that the peak message rate from a data source is 5000 per second, each message is 1 KB, the user population is 10 million, and the average query selectivity is as low as 0.001%. A back-of-the-envelope calculation gives an estimation of the result volume as 4 Gb per second. Disseminating this volume of data from a central server can be prohibitively expensive.

Having outlined the problem of large-scale XML-based data dissemination, we next present the position of our work within the large body of related work.

2.3 Related Systems

Publish/subscribe systems such as TIBCO Rendezvous [29], Gryphon [1][4], and Siena [9][10] provide distributed subject/content-based data dissemination. Distributed processing spreads the processing load and has the potential of scaling up for both service inputs and outputs. These systems, however, support limited expressiveness in message filtering. Earlier Publish/subscribe systems are subject-based [29]. In such systems, publishers label each message with a subject from a pre-defined set, and users subscribe to all the messages in a specific subject. The expressiveness of this service is restricted by the opaqueness of the message content in its data model. More recent publish/subscribe systems model messages as attribute-value pairs, and allow user profiles to contain a set of predicates over the values of those attributes [1][9][10][19][30]. The expressiveness of these systems amounts to filtering tuple-like messages based on the constituent attributes. Combining low expressiveness and high scalability, distributed pub/sub systems are represented by the upper left corner of the matrix shown in Figure 2.

More recently, a large number of XML filtering approaches have been developed [2][8][12][16][20][22][26][38]. These approaches typically support a subset of XPath 1.0 [15]. XML filtering provides more expressiveness in specifying data interests, resulting in more accurate filtering of messages. YFilter [17], a multi-query processor that we built previously, also supports result customization using a subset of XQuery. Although these XML filtering and transformation systems provide higher levels of expressiveness, their centralized style of processing limits their scalability. Revisiting Figure 2, today's XML filtering and transformation systems can be best described by the lower right corner of the matrix combining lower scalability and higher expressiveness.

Our work on content-based data dissemination adopts the paradigm of distributed processing to exploit aggregated bandwidth and processing power. As indicated in Figure 2, our system ONYX incorporates the high level of expressiveness of XML filtering and transformation into a distributed data dissemination service.

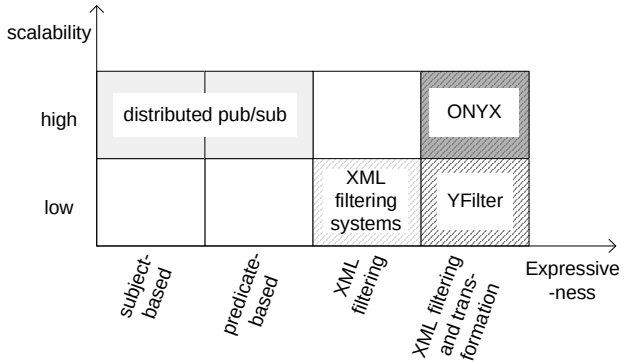

Fig. 2. Combining expressiveness and scalability

3 System Model

In this section, we present the operational features of ONYX. ONYX provides content-based many-to-many data dissemination from publishers to end users. It consists of an overlay network of nodes. Most of the nodes serve as information brokers (or brokers, for short) that handle messages and user queries, while a few of them collaborate to provide a registration service. The overview is illustrated in Figure 3.

3.1 Service Interface

The service interface provided by ONYX consists of several methods (some of which are similar to those in [3]):

Register a data source: A data source registers with ONYX by contacting the registration service and providing information about its location, the schema used, the expected message rate and message size, etc. (as illustrated by message 1 in Figure 3). The registration service assigns an ID to the data source, and chooses a broker as the *root broker* for the data source. The choice of the root broker is based on its topological distance to the data source, the bandwidth available, and the data volume expected from that source. After the service forwards the information about the new data source to the root broker (message 2), it returns the assigned ID and the address of the root broker to the data source (message 3).

Publish data: After registration, a data source publishes its data by attaching its ID to each message and pushing the message to its root broker (message 4).

Register a data interest: To subscribe, the user contacts the registration service, and provides his profile and network address (message 5). The registration service assigns an ID to this profile, and chooses a broker as the *host broker* for this profile based on the user's location and/or the content of the profile. At the end of the registration, the service forwards the profile and related information to the host broker (message 6), and returns the profile ID and host broker address to the user (message 7). Thereafter, the host broker will deal with all the user requests concerning that profile.

Update a data interest: Subsequent changes to a profile (including updates and deletion) are sent directly to the host broker (message 8).

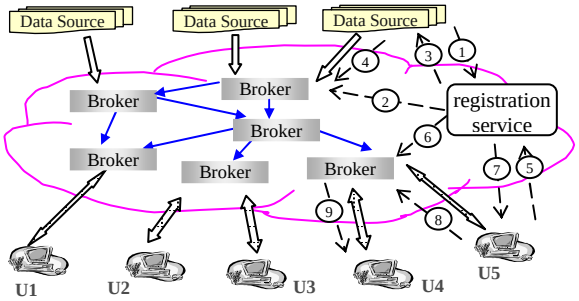

Fig. 3. Architecture of ONYX

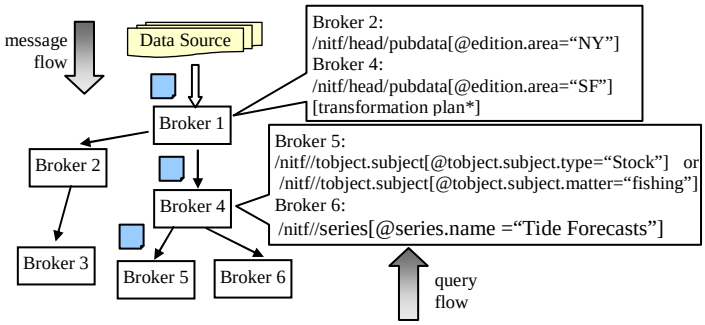

Fig. 4. Message routing based on content

Note that users do not need a method to retrieve the messages matching their interests, because those messages are pushed to them from the system (e.g., message 9). Additional methods are provided for data sources to update the schema and other information sent previously.

Fault-tolerance can be achieved by having backup nodes for the registration service and the brokers or using other techniques. That discussion is beyond the scope of this paper.

3.2 Two Planes of Content-Based Processing

ONYX is an application-level overlay network. It consists of two layers of functionality. The lower layer, called the *control plane*, deals with application-level broadcast trees and gives each broker a broadcast tree rooted at that broker that reaches all other brokers in the network. Figure 4 shows such a tree in a network consisting of six brokers. Algorithms for constructing broadcast trees have been provided elsewhere (e.g., [14]).

In this section, we focus on the higher layer of functionality in ONYX – c*ontent-based processing*, which is the primary concern of this paper. We decompose the operations in this layer into two planes of processing - the *data plane* and the *query plane.* The data plane captures the flow of messages in the system while the query plane captures the flow of queries and query-related updates in the system. As we will see, the duality of data and query is a pervasive feature of ONYX. We now discuss the three tasks performed in this layer – content-driven routing, incremental transformation, and user query processing.

Content-driven routing is necessary to avoid the flooding of messages to all brokers in the network. It builds on top of the broadcast tree described above. The routing is *content-driven* because instead of forwarding a message to all the children in the broadcast tree, a broker sends it to only the subset that is "interested" in the message. This routing scheme, which matches a message's content with routing table entries (or *routing queries*) representing the interests of child brokers, is in sharp contrast to the address-based IP routing scheme.

Figure 4 shows an example of routing a message based on its content. The routing tables for Broker 1 and 4 are shown conceptually. The table at Broker 1 specifies a routing query "*/nitf/head/pubdata*[*@edition.area*= "*NY*"]" for Broker 2, and a similar one "*/nitf/head/pubdata*[*@edition.area*= "*SF*"]" for Broker 4. The matching of a new message arriving at Broker 1 with either routing query results in routing the message to the corresponding child. The building of such routing tables by summarizing the queries of downstream brokers is a subtask in the query plane. The matching of messages against routing queries occurs in the data plane.

Incremental transformation is the second task in the content-based processing layer. Interesting cases of transforming messages during routing include (1) early projection, i.e., removal of data, and (2) early restructuring. An example of early projection is as follows. A data source publishes messages containing multiple news articles. If all the user queries downstream of a link are interested only in a subset of the articles (e.g., those distributed in the area "SF"), messages can be projected onto the articles of interest before they are forwarded along that link using the following query:

```
<batched-nitf>
{ for $n   in    $msg/batched-nitf/nitf
  where          $n/head/pubdata/@edition.area ="SF"
  return         $n
}
</batched-nitf>
```

An example of restructuring is message transcoding based on the profiles of wireless users, say, when all users downstream of a link require images and comments to be removed and tables to be converted to lists. Incremental transformation helps reduce message sizes and avoids repeated work at multiple brokers.

We enable incremental transformation by attaching transformation queries to the output links of brokers on the path of routing. User queries downstream of a link are aggregated and the commonality in their transformation requirements is extracted to form the transformation query. These subtasks happen in the query plane. The corresponding subtask in the data plane consists of transforming messages using these queries, before the messages are sent to the output links.

User query processing is the task of matching and transforming messages against individual user queries at their host brokers. For the user queries resident at a particular broker, this is the last step of message processing (although the arriving messages may be routed and transformed for other downstream user queries). The subtask in the query plane consists of issues such as indexing of user queries for which the broker is a host broker, and the subtask in the data plane consists of matching messages against these indexes.

Table 1 summarizes the content-based processing tasks in ONYX and their subtasks over the query and data planes.

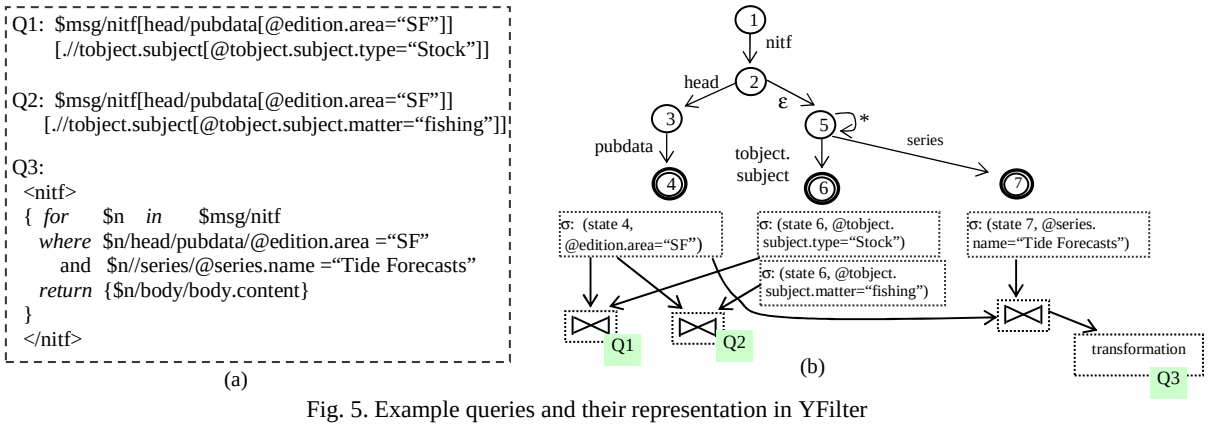

Fig. 5. Example queries and their representation in YFilter

System Task	Query Plane	Data Plane
Content-driven routing	build routing tables	lookup in routing tables
Incremental transformation	build transformation plans	execute transformation plans
User query processing	build query plans	execute query plans

Table 1: System tasks over the two planes of processing

4 Core Techniques

In this section, we describe three key aspects of ONYX, the query plane, the data plane, and the query partitioning strategy. YFilter serves as a basis for these components, so we first present some YFilter basics.

4.1 YFilter Basics

YFilter [16][17] is an XML filtering and transformation engine that processes multiple queries in a shared fashion. In the core of YFilter, a *Non-Deterministic Finite Automaton* (NFA) is used to represent a set of simple linear paths and support prefix sharing among those paths. YFilter provides a fast algorithm for running the NFA on an input message to match the contained paths simultaneously, and an incremental approach for maintaining the NFA when some of the paths change.

While the structural components of path expressions are handled by the NFA, for the remaining portions of the queries, YFilter builds a network of operators starting from the accepting states of the NFA. Each operator performs a specific task, such as evaluation of value-based predicates, evaluation of nested paths, or transformation. The operators residing at an accepting state of the NFA can be executed when that accepting state is reached. Downstream operators in the network are activated when all their preceding operators are finished. In addition, some accepting states and operators are annotated with query identifiers. These identifiers specify that if an annotated accepting state is reached or an annotated operator is successfully evaluated, the queries corresponding to the identifiers are satisfied.

Figure 5 shows three example queries and their representation in YFilter. Take Q1 for example. It contains a root element "/nitf" with two nested paths applied to it. YFilter decomposes the query into two linear paths "/nitf/head/pubdata[@edition.area="SF"]", and "/nitf//tobject.subject[@tobject.subject.type="Stock"]". The structural part of these paths is represented using the NFA (see Figure 5(b)), with the common prefix "/nitf" shared between the two paths. The accepting states of these paths are state 4 and state 6, where the network of operators (represented as boxes) for the remainder of Q1 starts. At the bottom of the network, there is a selection (σ) operator below each accepting state to handle the value-based predicate in the corresponding path. For example, the box below state 4 specifies that the predicate on the attribute *edition.area* should be evaluated against the element that drove the transition to state 4. To handle the correlation between the two paths (e.g., the requirement that it should be the same *nitf* element that makes these two paths evaluate to true), YFilter applies a join (⊳⊲) operator after the two selections. This operator realizes the correct semantics of the nested paths. In Figure 5(b), the left most join operator is annotated with the query identifier Q1. This means that if the join is successfully evaluated, then Q1 is satisfied.

The representation of Q2 follows the same two paths in the NFA as Q1 and uses the same selection at state 4 to process the common predicate with Q1, but it contains a separate selection at state 6 to evaluate the different predicate in the second path. A distinct join operator is built on these two selections. The representation of Q3 is similar to that of Q1 and Q2 for the *for* and *where* clauses, but contains an additional box for transformation using the *return* clause. For more details on YFilter, the interested reader is referred to [16][17].

4.2 Query Plane

In this subsection, we focus on two issues on the query plane: routing table construction and the generation of incremental transformation plans. Our solutions are based on an extension of the YFilter processor. Note that we do not discuss user query processing, as it is completely handled by YFilter.

4.2.1 Routing Table Construction

As stated previously, a routing table conceptually consists of routing query-output link pairs, where each routing query is aggregated from user queries downstream of the corresponding output link. In our work, we decided to implement routing tables using YFilter for three reasons: (1) fast structure matching of path expressions using the NFA, (2) the small maintenance cost of an NFA for query updates (e.g., com-

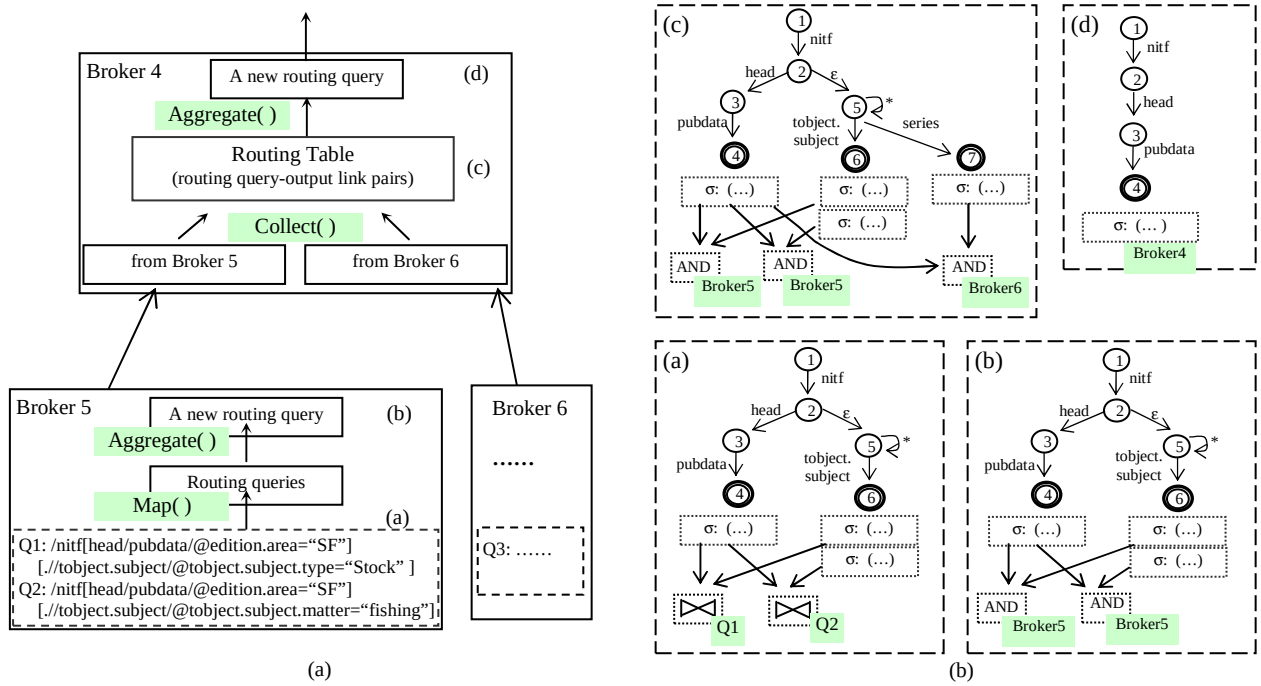

Fig. 6. Examples of constructing routing tables using a disjunctive normal form

pared to deterministic automata), and (3) extensibility for supporting new operations using operator networks. Here, we present the representation of routing tables and mechanisms to construct them. For the purpose of routing, we only consider the matching part of a query, i.e., the *for* and *where* clauses of a query written in XQuery. This part can be converted to a single path expression with equivalent semantics, which we refer to as the *matching path* of a query.

In our current design, routing queries are represented using a *Disjunctive Normal Form* (DNF) of absolute linear path expressions. If a matching path contains n nested paths, it is decomposed into n+1 absolute linear paths (possibly with value-based predicates). The routing query constructed for this matching path is the conjunction of the resulting n+1 paths. Multiple routing queries can be connected using *or* operators to create a new routing query. Note that an alternative could be to allow any matching path to be a routing query and use *or* operators to connect them. In comparison, DNF relaxes the semantics of nested paths. The motivation of using DNF is that join operators used to evaluate nested paths are relatively expensive, whereas logical *and* operators between path expressions can be evaluated much more efficiently. Investigation of alternative forms is one direction of our future work.

Routing table construction from a distributed query population consists of applying three functions, *Map*( ), *Collect*( ), and *Aggregate*( ), to create routing queries in the chosen form.

- *Map*( ) maps the matching path of a user query to the canonical form of a routing query;
- *Collect*( ) gathers routing queries sent from the child brokers into the routing table of a broker;
- *Aggregate*( ) merges the routing queries in the routing table of a broker with those mapped from the user queries at the broker, and generates a new routing query to represent the broker in its parent broker.

These three functions are illustrated for Brokers 4 and 5 in Figure 6(a). Broker 5 is a host broker with matching paths Q1 and Q2. It uses function *Map*( ) to create a routing query for each of them. Then it applies *Aggregate*( ) to those routing queries to generate a new one that will represent it in its parent (Broker 4). Note that as a leaf, Broker 5 does not contain a routing table. Broker 4 has child brokers Broker 5 and Broker 6, but no user queries. It uses function *Collect*( ) to merge the routing queries sent from the child brokers into a routing table, and then applies *Aggregate*( ) to the routing table to generate a routing query that will represent it in its parent.

Construction operations. Next we present the implementation of the three functions using YFilter.

Map( ) takes as input a YFilter operator network representing a set of matching paths. To create the DNF representations of their routing queries, *Map*( ) simply replaces each join operator in the operator network with an *and* operator.

Collect( ) merges routing queries sent from downstream brokers into a routing table of a parent broker. This operation simply merges the YFilter operator networks that represent those routing queries.

Aggregate( ) performs re-labeling on a YFilter operator network. It changes all the identifier annotations (for queries or brokers) to the identifier of this broker, so that the annotated places become marks for routing to this broker. It essentially adds "*or*" semantics to those annotated places, as encountering any one of them can cause routing of messages to this broker. YFilter treats broker identifiers the same as query identifiers, so these identifiers are simply called "targets" in the sequel.

An example is shown in Figure 6(b). Box (a) in this figure shows the YFilter operator network built for queries Q1

and Q2 from Broker 5. Box (b) represents the routing query created for Broker 5 after applying *Map*( ) and *Aggregate*( ) to box (a). Box (c) depicts the result of merging box (b) with the routing query sent from Broker 6 (assumed to be the routing query created for query Q3 in Figure 5(a)). Box (d), the result of applying *Aggregate*( ) to box (c), will be explained shortly below.

Sharing among routing queries. It is important to note the difference between the conceptual representation of a routing table (i.e., routing query-output link pairs) and our implementation of it. Instead of creating a separate operator network for each routing query, we represent all the routing queries in a routing table using a single combined operator network. As a result, the common portions of the routing queries will be processed only once. As an example, box (c) in Figure 6(b) shows that the path leading to accepting state 4 and the selection operator attached to that state can be shared between the routing query for Broker 5 and that for Broker 6. When the commonality among routing queries is significant, the benefit of sharing can be tremendous.

The *or* semantics introduced to routing queries, however, complicates the issue of sharing. When using separate operator networks for routing queries, a short-cut evaluation strategy can be applied in the evaluation of each routing query. Consider box (b) in Figure 6(b) as an operator network created for the routing query for Broker 5. If during execution, one of the two targets labeled as Broker 5 is encountered, the processing for this routing query can stop immediately. In contrast, when using the combined operator network shown in box (c), after a target for Broker 5 is encountered, the processing of the combined operator network has to continue as the target for Broker 6 has not been reached. If care is not taken, some future work may be performed which only leads to the targets for Broker 5. In other words, naïve ways of executing a combined operator network for shared processing may perform wasteful work.

To solve this problem, our solution is to have a runtime mechanism that instructs YFilter to ignore the processing for duplicate targets but not the processing for different targets. This mechanism is based on a dynamic analysis of the operator network which reports the portions of the combined operator network that will only lead to the targets that have already been reached.

Content generalization. Another issue to address in routing table construction is the size of routing tables (i.e., the size of their operator network representation). Larger routing tables can incur high overhead for routing table lookup, thus slowing the critical path of message routing. They may also cause memory problems in environments with scarce memory. For these reasons, we introduce *content generalization* as an additional step that can be performed in *Collect*( ) or *Aggregate*( ). Generalizing the routing table essentially trades the filtering power of the routing table for processing or space efficiency.

We propose an initial set of methods for content generalization. Some of methods generalize individual path expressions with respect to their structural or value-based constraints. Some other methods generalize all the disjuncts in a routing query. For instance, one such method preserves only the path expressions common to all the disjuncts in the new routing query. Consider the routing table shown in box (c) in Figure 6(b). When applying *Aggregate*( ) to this routing table, calling this method after re-labelling the identifiers will result in an operator network containing a single path, as shown in box (d). This generalized operator network will be used to represent Broker 4 in its parent.

4.2.2 Incremental Message Transformation

Incremental transformation happens in the course of routing. As mentioned in Section 3, it can be an early projection or an early restructuring. In this subsection, we briefly describe the extraction of incremental transformation queries from user queries and the placement of these transformation queries.

A transformation query for early projection can be attached to an output link at a broker, if (1) its *for* clause is shared by all the user queries downstream of the link, (2) its *where* clause generalises the *where* clauses of all those queries, and (3) the binding of its *for* clause provides all the information that the *return* clauses of those queries require. The last requirement implies that the *return* clauses of the user queries downstream cannot contain absolute paths or the backward axis ".." to navigate outside the binding.

Similarly, a transformation query for early restructuring can be applied to an output link, if conditions (1) and (2) above are satisfied, and (3) the *return* clauses of the downstream queries all contain a series of transformation steps (e.g., removing images and then converting tables to lists), and the first few steps are shared among all those queries. This transformation query will carry out the common transformation steps on matching messages earlier at this broker.

When opportunities for early transformation are identified at host brokers based on the above conditions, incremental transformation queries representing them are generated and propagated to the parent broker. At the parent, these transformation queries are compared and the commonality among them is extracted to create a new transformation query for its own parent and a set of "remainder queries" for its output links. A remainder query is one that combined with the new transformation query constitutes the original transformation query. Each remainder query is attached to the output link where the corresponding original transformation query came from. The new transformation query is propagated up, and the above process repeats.

A final remark is that although our algorithms for routing table construction and incremental transformation plan construction as presented consider all the user queries in a batch, they can also be applied for incremental maintenance of routing tables or transformation plans. In that case, "delta" routing/transformation queries are constructed and propagated, instead. Details are omitted here due to space constraints.

4.3 Data Plane

Having described the query plane, we now turn to the data plane that handles the XML message flow. In the following, we describe two aspects of this plane, holistic message processing for various tasks and efficient XML transmission.

4.3.1 Holistic Message Processing

In ONYX, a single YFilter instance is used at each broker to build a shared, "holistic" execution plan for the routing table,

incremental transformation queries, and local user queries (by holistic, we mean that all these processing tasks are considered as a whole in the data plane). Processing of an XML message using this shared plan is sketched in this section.

The execution algorithm for holistic message processing is an extension of the push-based YFilter execution algorithm [17]. As in that previous work, elements from an XML message are used to drive the execution of NFA. At an accepting state of the NFA, path tuples are created and passed to the operators associated with the state. The network of operators is executed from such operators (i.e., right below accepting states) to their downstream operators. In YFilter, the order of operator execution is based on a FCFS policy among the operators whose upstream operators have all been completed.

In contrast to earlier work, however, the holistic plan contains multiple types of queries, i.e., routing queries, incremental transformation queries, and local user queries. The first two types are on the critical path of message routing. They should not be delayed by the processing for local queries. Moreover, incremental transformation is useful only if the routing query for the corresponding link can be satisfied, which implies the dependency of transformation queries on the routing queries in execution. For these reasons, we propose a dependency-aware priority scheduling algorithm to support shared holistic message processing.

Dependency-aware priority scheduling. In this algorithm, operators that contribute to routing queries are assigned high priority; among other operators, those that contribute to incremental transformation queries have medium priority; and the rest of the operators have low priority. The second priority class, however, is declared to be dependent on the first class with the following condition: an operator in the second class is executed only if at least one incremental transformation query that it contributes to has been necessitated by the successful evaluation of the corresponding routing query. In our implementation, an FCFS queue is assigned to each priority class. In addition, a wait queue is assigned to the dependent class. Priority scheduling works as in a typical OS, except that operators in the dependent class are first placed in the wait queue, and then moved to the FCFS queue when their dependency conditions have been satisfied.

4.3.2 Efficient XML Transmission

Low cost transmission of XML messages is also a paramount concern in a multi-hop distributed dissemination system. XML raises two challenges in this context. First, the verbose nature of XML can cause many redundant bytes in the messages. Second, XML messages need to be parsed at each broker, which can be expensive [16][36]. In this section, we address these two challenges.

The inherent verbosity of XML has led to compression algorithms such as XMill [27]. Compression, however, solves only the first of the above challenges but not the parsing problem. A promising approach that we explored to counter this problem, is using an *element stream* format for XML transmission. This format is an in-memory binary representation of XML messages that can be input to the YFilter processor without any pre-processing or parsing. The binary format is also more space-efficient than raw XML because the latter has white spaces and delimiters. The "wire size" of an XML message can be further reduced by compressing this binary representation.

We also explore schema-aware representation of XML for transmission. Given that the control plane can be used to broadcast the schema of a publishing source to all the brokers in the network, we can perform schema-aware XML encoding of messages for transmission between brokers. In particular, we use a dictionary encoding scheme that maps XML element and attribute names from the schema to a more space-efficient key space. As future work, we would like to explore more advanced schema-aware optimizations, such as avoiding storing parent-child relationships in the binary format, as they can be recovered from the schema.

We experimented with six XML transmission formats: text, binary (i.e., the element stream format), binary with dictionary encoding, and their corresponding compressed versions. Messages were generated using the YFilter XML Generator [16] based on the NITF DTD. The two parameters - *DocDepth* (that bounds the depth of element nesting in the message) and *MaxRepeats* (that determines the number of times an element can repeat in its parent element) allow us to vary the complexity of messages. All our compression was performed using ZLIB, gzip's library, because it outperforms XMill for the relatively small-sized messages (like ours), as reported in [27].

Figure 7 summarizes the performance of different XML formats over our first metric, the *wire size*, for messages of different complexities. Although the element stream format does not remarkably outperform the text format, dictionary encoding gives promising results. Compression helps reduce the wire size for all formats significantly.

Figure 8 presents the evaluation of these XML formats on the complementary metric of *message processing delay*. While uncompressed formats require only serializing messages at the sender and deserializing them at the receiver, the raw format additionally requires parsing and thus proves to be expensive. Compressed formats have significant costs of compression at the sender and decompression at the receiver.

The choice of XML format for transmission must weigh both the wire size and processing delay metrics to get a combined metric. This decision will invariably be influenced by implementation details like the transport protocol used. For example, in the distributed PlanetLab testbed [31], all the message sizes involved in our experiments gave the same transmission delay using TCP. This was attributed to the connection establishment time dominating in TCP for small message sizes. Thus, the message processing delay turned out to be a more important concern than the message size, making compression rather undesirable. On the other hand, if the DCP protocol [36] that sends data in redundant streams over UDP can be employed, compression may be useful.

4.4 Query Population Partitioning

Previous work on distributed publish/subscribe [1][4][10] assumes that queries naturally reside on their nearest brokers, without considering alternative schemes for partitioning the query population. In this subsection, we address the effect of query partitioning on the filtering power of content-driven routing, which is captured by the fraction of query partitions that a message can match.

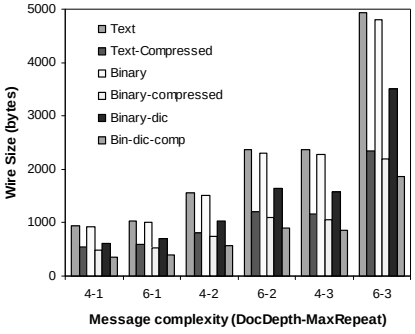

Fig. 7. Wire size of XML messages

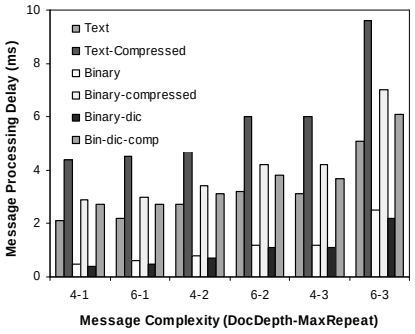

Fig. 8. Processing delay for XML transmission

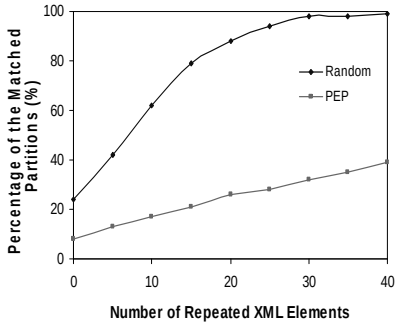

Fig. 9. Random query partitioning vs. PEP

We start with an investigation of the properties of query partitioning and their effect on content-driven routing. Query similarity within a partition seems to be an intuitive property, but is not effective in filtering. For example, in the ideal case that all the queries in one partition are "/a/b" and all the queries in the other partition are "/a/c", a message can still match both partitions by containing the two required elements. Dissimilarity between partitions is another candidate. Consider one partition with two queries "//a" and "//b", and the other partition with "//c" and "//d". Though these two partitions have little in common, it is still quite likely that a message matches both partitions. Mutual exclusiveness turns out to be a desired property. For example, if one partition requires "/a/b[@id=1]" and the other requests "/a/b[@id=2]", the chance that a message satisfies both can be low. The message surely cannot satisfy both if it contains only one "b" element.

The next question is what path expressions can establish such mutual exclusiveness among query partitions. In this regard, we make three key observations. The first is that structural constraints alone are not enough (see the first two examples above). This is because the schema never specifies that two paths are mutually exclusive in a message. In fact, path expressions exhibit potential exclusiveness if they involve the same structure, and contain value-based predicates that address the same target (e.g., an attribute or the data of a specific element), use the "=" operator, but contain different values (see the third example above). We call the common part of these paths an *exclusiveness pattern*. The second observation is that repetition of element names in XML messages limits the exclusiveness of such patterns. Thus, the best choice of an exclusiveness pattern would be one that can appear at most once in any message, as dictated by the schema. The third observation is that in general the coverage of an exclusiveness pattern in the query population could be rather limited, due to the diversity of user data interests. Thus, using a single exclusiveness pattern for query partitioning could cause the majority of queries to be placed in a partition called "don't care". In that case, a set of exclusiveness patterns should be used.

Partitioning based on Exclusiveness Patterns. To achieve exclusiveness of data interests among query partitions, we propose a query partitioning scheme, called *Partitioning based on Exclusiveness Patterns* (PEP). Due to space constraints, we only briefly describe the two steps of this scheme, assuming for now that this algorithm can be run over the entire query population in a centralized fashion. (1) *Identifying a set of exclusiveness patterns*. PEP first searches the YFilter representation of the entire query population, and aggregates the predicates contained in the selection operators at each accepting state to exclusiveness patterns. These patterns are sorted by their coverage of the query population (i.e., the number of queries involving them). Then PEP uses a greedy algorithm to choose a set of patterns such that every query involves at least one pattern from the set. Heuristics can be used to perturb this set with other unselected patterns so that more patterns included in the set can appear at most once in a message, but the coverage of the query population is not sacrificed. (2) *Partition creation.* In the second step, K query partitions are created using the M patterns selected in the first step. To do so, the value range of each exclusiveness pattern is partitioned into K buckets, numbering 1, 2, …, K. Then queries are assigned to the $K*M$ buckets based on their values in the contained exclusiveness patterns. As a query must involve at least one of those patterns, it must belong to at least one bucket. If the query involves multiple patterns, it is randomly assigned to one of the matching buckets. Finally, K query partitions are created by assigning the queries in the i^{th} bucket of any pattern to query partition i.

In the ideal case, where each exclusiveness pattern appears at most once in a message, a message can match at most M query partitions, i.e., one bucket per pattern. Thus the filtering power of content-driven routing, i.e., the fraction of query partitions that a message can match, can achieve M/K (e.g., 10 patterns, 100 partitions, and filtering power ≈ 1/10). If some patterns can appear multiple times in a message, their repetition degrades the filtering power (in many cases linearly).

To study the potential benefit of our PEP scheme, we compared its performance with the random query partitioning scheme that randomly assigns queries to partitions. We considered assigning a population of 1 million queries to 200 partitions. Every query contained two patterns, each chosen uniformly from a set of 10 exclusiveness patterns. PEP exploited these 10 patterns for partitioning. Figure 9 shows how the percentage of the partitions that a random message matches varies with the amount of repetition of element names in the XML message. Clearly, the random partitioning scheme ends up matching almost all partitions with messages even with a small amount of repetition of element names. In contrast, PEP leads to many fewer partition matches. Unless user interests are influenced by geography, a system that assigns user queries to the closest brokers will end up doing

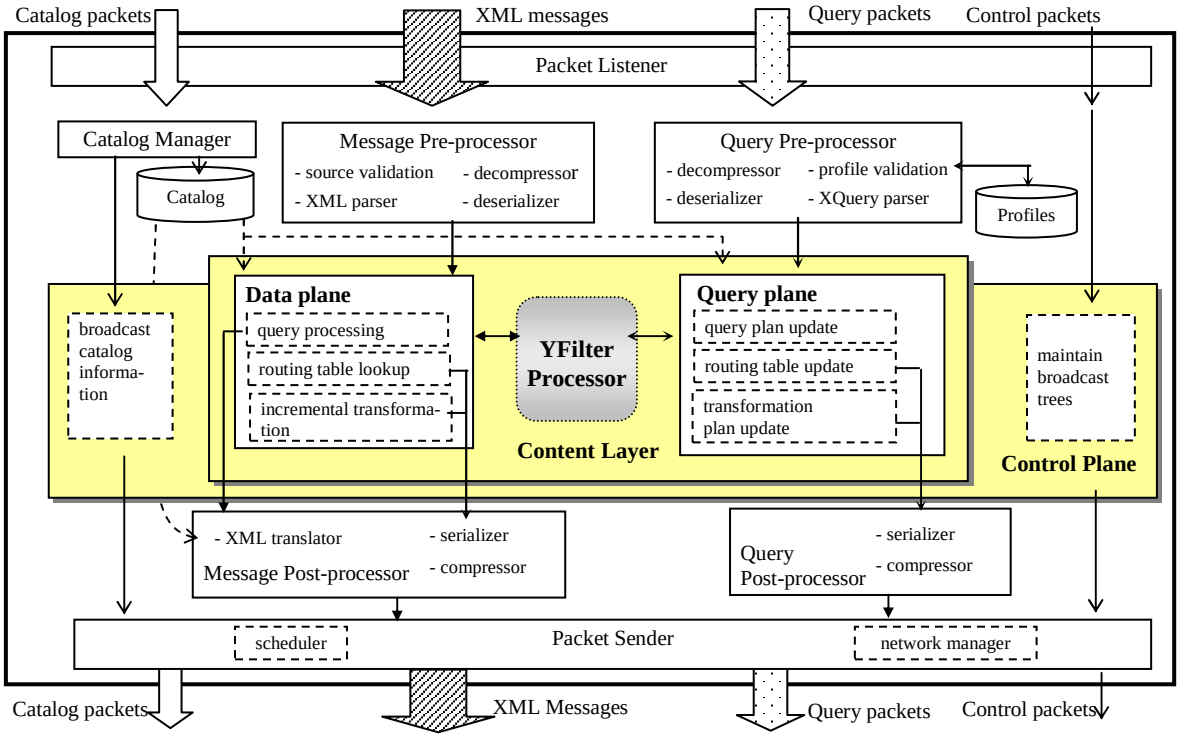

Fig. 10. Broker Architecture

random partitioning of queries, leading to many messages being exchanged between the brokers of the system.

An important remark is that in ONYX, PEP is a core algorithm for query placement used by the registration service. In addition to PEP, query placement also involves the decision of mapping query partitions to brokers, and the use of distributed protocols to perform the initial query partitioning and to maintain the partitions as user queries change over time. These issues will be addressed in our future work.

5 Broker Architecture

Having described the broker functionality in the query and data planes, we now turn to a discussion of the broker architecture that implements this functionality. This architecture is shown in Figure 10. It contains the following components.

Packet Listener. This component listens to each packet arriving at the broker and based on the header, assigns the packet to one of the four flows: catalog packets, XML messages, query packets, and network control packets.

Catalog manager. Catalog packets contain information about a data source. They may originate from the registration service concerning a new data source or from a registered data source to update information sent previously. The catalog manager parses these packets, and stores the information in the local catalog. If the packet is for a new data source, a new entry is added to the catalog including the ID of the data source, information on the data rate, the schema used, etc. If the information relates to a known data source, the existing entry in the catalog describing this data source is updated by the new information. The catalog will be used in other components for message validation, XML formatting, query processing, etc.

Message pre-processor. XML messages can come from data sources as well as other brokers in the system. The messages from a data source carry the source ID and are in the text format. On receiving such a message, the root broker of the data source validates the source ID attached to the message using its catalog. It also parses the message to an in-memory representation for later routing and query processing. If the message comes from an internal broker, source validation is skipped. Depending on the internal representation of XML, the message can be in one of several formats that we discussed earlier, and will need suitable pre-processing (like decompression, deserialization, etc.).

Query pre-processor. This is analogous to the message pre-processor in functionality, except that it also maintains a database of the profiles for which it is the host broker.

Control plane: Taking the control messages, the control plane maintains the broadcast tree for each root broker in the system. Specifically, it records the parent node and the child nodes of a broker on a particular root broker's broadcast tree. It provides two methods for use of the content layer, one for forwarding messages along a broadcast tree, the other for reverse forwarding of queries. The control plane is also responsible for disseminating catalog information for the purposes of optimizing content-based processing. For example, the schema information can be used to optimize query processing and support schema-aware XML encoding.

Data plane. The broker performs three tasks in the data plane, when receiving an XML message. First, it takes a sequence of steps to route the message: (a) if the broker is the root broker for the message, it attaches its broker identifier to

the message; (b) it retrieves its output links in the broadcast tree that is specified by the root broker identifier attached to the message; and (c) it looks up in the content-based routing table to filter those output links. Second, for each output link selected, the broker transforms the message, if a transformation plan is attached to that link. Last, the broker processes the message on local queries to generate results. These three tasks are all realized by the YFilter processor.

Query plane. The query plane exhibits duality with the data plane. If an arriving query is from a user, the local query processing plan is updated. If the query comes from another broker to update the routing table (i.e., it is a routing query) or the incremental transformation plan (i.e., it is an incremental transformation query), the modification of the routing table or the transformation plan will cause a new query to be generated for delivery to its parent broker.

YFilter Processor. YFilter has been described in Section 4.1. In this work, it is leveraged to build a holistic processing plan for all the processing tasks, so that the shared processing among the tasks is maximized. For the query plane, it is extended to support the routing table construction operations (as described in Section 4.2.1). For the data plane, its scheduler is augmented to prioritize the processing for different types of queries while exploiting the sharing among them (see Section 4.3.1).

Message and query Post-processor. The results from the data plane are passed to the message post-processor. Results of local query processing are translated into XML messages for delivery to end users, while results of routing and incremental transformation are serialized (and possibly compressed). Queries generated from the query plane also follow the path of serialization and compression.

Packet Sender. This component attaches a header to each packet, specifying the type of flow, the identifier of the root broker (if the packet is an XML message), and the format used. Then it multiplexes the four types of flows into the output channel, through a scheduler and a network manager that sends packets through TCP, UDP, etc.

6 Related Work

Our work is related to a large body of research work in both database and networking communities. Some areas like XML filtering have been described in detail already; we now present a brief overview of other related work.

Multicast. Multicast allows a source to send the same content to multiple receivers. Though bandwidth-efficient, IP multicast [24] is not flexible because of being a network layer paradigm. This has led to application-layer solutions such as *Overcast* [25] and *i3* [37]. Proposals for augmenting IP multicast with content-based routing features have been presented in [35][30]. However, none of this work gives the user fine-grained ways of specifying their interests, like a powerful query language over XML.

Content Distribution Networks (CDN). CDNs provide an infrastructure that delivers static or dynamic Web objects to clients from nearby Web caches or data replicas [13][40], thus offloading the main website. Recent work has focused on allowing the user to specify coherence requirements over data [1][34]. This differs from our approach as it does not give the user a powerful query language to specify her interests. Also, we are dealing with streams of XML messages rather than Web objects.

Publish/Subscribe systems. Publish/Subscribe systems are event-based and provide many-to-many communication between event publishers and subscribers. The SIFT system [41] provided support for matching keyword queries over large sets of documents and some ideas for building a distributed filtering system. Many recent systems [1][9][10][19][30] model an event as a conjunction of (attribute, value) pairs and support relational predicates in subscriptions specifying event interests. We are addressing a more challenging problem as support for rich XML messages and queries leads to increased complexity of query processing, data forwarding and routing table construction.

XML-based overlay networks. A mesh-based overlay network has been proposed in [36] with support for simple XML queries. However, the authors do not address XML query processing issues. The query aggregation scheme given in [11] has been used to perform content-based routing in [13]. However, they do not support powerful query language features like customized transformations.

Transcoding. The transformation functionality in our system is closely related to the transcoding of Web content to suit the profiles of heterogeneous end users, like the users of mobile phones and hand-held computers [42]. However, such a profile usually does not provide expressiveness in querying content as much as the subset of XQuery we support.

7 Status and Future Work

In this paper, we presented our initial design of ONYX, a distributed system providing large-scale XML dissemination. In particular, we provided a detailed architectural design of the system, and addressed the various challenges in distributed XML dissemination in the context of leveraging YFilter, a state-of-the-art XML processor. While we view this work as an initial step towards Internet-scale XML dissemination services, the proposed architecture and solutions to critical issues such as routing table construction and query population partitioning lay the foundation for offering high expressiveness and scalability in such services in massively distributed environments.

As of June 2004, we have implemented the components for message/query pre-processing and post-processing. A collaboration with the Berkeley networking group to build the networking related components, such as the control plane, is underway. We expect to fully implement the data and query planes using YFilter over the course of the summer, and deploy our system on PlanetLab [31] in the fall.

We also plan to extend our research work in the following directions. We will explore alternative forms of routing query representation in addition to DNF and other content generalization algorithms. Typical workloads of XML routing will be collected to evaluate these alternative forms and algorithms to gain insights into the various tradeoffs. We will also exploit the schema for optimization in routing table construction. Furthermore, we plan to extend the notion of data/query duality in the context of multi-source routing; analogous to placing routing queries to filter and direct the

message flow, we can place data source descriptions in the network to prune and forward the query flow from host brokers to root brokers. Last, we will address the networking issues that occur when using PEP to move queries away from their closest brokers, and provide distributed protocols to carry out PEP and to maintain the quality of query partitioning as user queries change over time.

8 References

[1] Aguilera, M.K., Strom, R.E., Sturman, D.C., Astley, M., and Chandra, T.D. Matching Events in a Content-Based Subscription System. In *Proc. of Principles of Distributed Computing* (*PODC'99*), May 1999.

[2] Altinel, M., and Franklin, M.J. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *VLDB 2000*, 53-64, Sep. 2000.

[3] Altinel, M., Aksoy, D., Baby, T., Franklin, M.J., Shapiro, W., and Zdonik, S.B. DBIS-Toolkit: Adapatable Middleware for Large Scale Data Delivery. In *SIGMOD 1999*, 544-546, 1999.

[4] Banavar, G., Chandra, T. D., Mukherjee, B., Nagarajarao, J., Strom, R. E., and Sturman, D. C. An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems. In *Proc. of the IEEE International Conference on Distributed Computing Systems* (ICDCS), 262-272, May 1999.

[5] Bell, T.C., Cleary, J.G., and Witten, I.H. *Text Compression*. Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[6] Bhide, M., Deolasse, P., Katker, A., Panchgupte, A., Ramamritham, K., and Shenoy, P. Adaptive Push Pull: Disseminating Dynamic Web Data. *IEEE Transactions on Computers*, 51(6), 652-668, May 2002.

[7] Boag, S., Chamberlin, D., Fernández, M.F., Florescu, D., Robie, J., and Siméon, J. XQuery 1.0: An XML Query Language. W3C Working Draft, Nov. 2003. http://www.w3.org/TR/xquery/.

[8] Bruno, N., Gravano, L., Doudas, N., and Srivastava, D., 2003. Navigation- vs. Index-based XML Multi-query processing. In *ICDE 2003*, 139-150, Mar. 2003.

[9] Carzaniga, A., Rutherford, M.J., and Wolf, A.L. A Routing Scheme for Content-Based Networking. In *Proc. of IEEE INFOCOM 2004*, Mar. 2004.

[10] Carzaniga, A., and Wolf, A.L. Forwarding in a Content-Based Network. In *SIGCOMM 2003*, 163-174, Aug. 2003.

[11] Chan, C.Y., Fan, W., Felber, P., Garofalakis, M.N., and Rastogi, R. Tree Pattern Aggregation for Scalable XML Data Dissemination. In *VLDB 2002*, Aug. 2002.

[12] Chan, C., Felber, P., Garofalakis, M., and Rastogi, R. Efficient Filtering of XML Documents with XPath Expressions. In *ICDE 2002*, 235-244, Feb. 2002.

[13] Chand, R., and Felber, P. A Scalable Protocol for Content-Based Routing in Overlay Networks. In *Proc. of the IEEE International Symposium on Network Computing and Applications* (*NCA'03*), Apr. 2003.

[14] Chu, Y. Rao, S.G., and Zhang, H. A Case for End System Multicast. In *Proc. of the 2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 1-12, Jun. 2000.

[15] Clark, J., and DeRose, S. XML Path Language (XPath) - Version 1.0. Online at http://www.w3.org/TR/xpath.

[16] Diao, Y., Altinel, M., Zhang, H., Franklin, M.J., and Fischer, P.M. Path Sharing and Predicate Evaluation for High-Performance XML Filtering. *TODS*, 28(4), 467-516, Dec. 2003.

[17] Diao, Y., and Franklin, M.J. Query Processing for High-Volume XML Message Brokering. In *VLDB 2003*, 261-272, Sep. 2003.

[18] Dilley, J., Maggs, B., Parikh, J., Prokop, H., Sitaraman, R., and Weihl, B. Globally Distributed Content Delivery. *IEEE Internet Computing*, 50-58, Sep.-Oct. 2002.

[19] Fabret, F., Jacobsen, H.A., Llirbat, F., Pereira, J., Ross, K.A., and Shasha, D. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe Systems. In *SIGMOD 2001*, 2001.

[20] Green, T. J., Miklau, G., Onizuka, M., Suciu, D. Processing XML Streams with Deterministic Automata. In *Proc. of Int'l Conf. on Database Theory* (*ICDT'03*), 173-189, Jan. 2003.

[21] Gryphon. http://www.research.ibm.com/gryphon/index.html.

[22] Gupta, A. K., and Suciu, D. Streaming processing of XPath queries with predicates. In *SIGMOD 2003*, Jun. 2003.

[23] Internal Press Telecommunications Council. News Industry Text Format. 2004. http://www.nitf.org/.

[24] Internet Protocol (IP) Multicast. *http://www.cisco.com/univercd/cc/ td/doc/cisintwk/ito_doc/ipmulti.htm.*

[25] Jannotti, J., Gifford, D.K., Johnson, K.L., Kaashoek, M.F., and O'Toole, J.W.Jr. Overcast: Reliable Multicasting with an Overlay Network. In *Proc. of the 4th Symposium on Operating System Design and Implementation (OSDI'00)*, Oct. 2000.

[26] Lakshmanan, L.V.S., and Sailaja, P. On Efficient Matching of Streaming XML Documents and Queries. In *EDBT 2002*, 142-160, Mar. 2002.

[27] Liefke, H., and Suciu, D. XMILL: An Efficient Compressor for XML Data. In *SIGMOD 2000*, 153-164, May, 2000.

[28] McCanne, S., Jacobson, V., Vetterli, M. Receiver0Driven Layered Multicast. In *SIGCOMM 1996*, 117-130, Aug. 2003.

[29] Oki, B., Pfleugl, M., Siegel, A., and Skeen, D. The Information Bus: an Architecture for Extensible Distributed System. In *SOSP 1993*, 58-68, Dec. 1993.

[30] Opyrchal, L., Astley, M., Auerbach, J., Banavar, G., Strom, R., and Sturman, D. Exploiting IP Multicast in Content-Based Publish-Subscribe Systems. In *Proc. of IFIP/ACM Int'l Conference on Distributed Systems Platforms*, 185-207, 2000.

[31] PlanetLab. http:// www.planet-lab.org.

[32] Rodriguez, P., Ross, K.W., and Biersack, E.W. Improving the WWW: Caching or Multicast? *Computer Networks and ISDN Systems*, 30(22-23,25), 2223-2243, Nov. 1998.

[33] Segall, B., Arnold, D., Boot, J., Henderson, M., and Phelps, T. Content Based Routing with Elvin4. In *Proc. of AUUG2K*, Canberra, Australia, Jun. 2000.

[34] Shah, S., Dharmarajan, S., and Ramamritham, K. An Efficient and resilient Approach to Filtering and Disseminating Streaming Data. In *VLDB 2003*, 57-68, Sep. 2003.

[35] Shah, R., Jain, R., and Anjum, R. Efficient Dissemination of Personalized Information Using Content-Based Multicast. In *Proc. of IEEE INFOCOM 2002*, Jun. 2002.

[36] Snoeren, A.C., Conley, K., and Gifford, D.K. Mesh-Based Content Routing using XML. In *SOSP 2001*, Oct. 2001.

[37] Stoica, I., Adkins, D., Zhuang, S., Shenker, S., and Surana, S. Internet Indirection Infrastructure. In *SIGCOMM 2002*, 73-88, Aug. 2002.

[38] Tian, F., DeWitt, D., Pirahesh, H., Reinwald, B., Mayr, T., and Myllymaki, J. Implementing a Scalable XML Publish / Subscribe System Using a Relational Database System. In *Proc. of SIGMOD 2004*, Jun. 2004.

[39] Tolani, P.M., and Haritsa, J.R. XGRIND: A Query-Friendly XML Compressor. In *ICDE 2002*, 225-234, Mar. 2002.

[40] WebSphere Application Server Network Deployment. http://www-306.ibm.com/software/webservers/appserv/was/network/edge.html.

[41] Yan, T. W., and Garcia-Molina, H. The SIFT Information Dissemination System. *TODS*, 24(4), 529-565, Dec. 1999.

[42] WebSphere Transcoding Publisher. http://www-306.ibm.com/software/pervasive/transcoding_publisher.

[43] NASDAQ Pre-Market Volume. http://dynamic.nasdaq.com/dynamic/premarket5dayvolume.stm.

[44] The NetLogger Toolkit. http://www-didc.lbl.gov/NetLogger/.