

Routing of XML and XPath Queries in Data Dissemination Networks *

Guoli Li

Shuang Hou

Hans-Arno Jacobsen

University of Toronto, Canada

Abstract

XML-based data dissemination networks are rapidly gaining momentum. In these networks XML content is routed from data producers to data consumers throughout an overlay network of content-based routers. Routing decisions are based on XPath expressions (XPE) stored at each router. To enable efficient routing, while keeping the routing state small, we introduce an advertisement-based routing algorithm for XML content, present a novel data structure for managing XPEs, especially apt for the hierarchical nature of XPEs and XML, and develop several optimizations for reducing the number of XPEs required to manage the routing state. The experimental evaluation shows that our algorithm and optimizations reduce routing table size by up to 90%, improve the routing time by roughly 85%, and reduce overall network traffic by about 35%.

1. Introduction

XML-based data dissemination networks are starting to become a reality. In a dissemination network, data, marked-up in XML, is routed based on filter expressions stored at intermediate nodes that indicate where the XML document is to be routed to. Filter expressions, often expressed as XPath expressions (XPEs), are submitted by data consumers who express interest in receiving certain kinds of documents. For instance, a globally operating insurance company with many branch offices distributed world-wide is linked by an overlay network of content-based routers that comprise the XML dissemination network. An insurance claim, an insurance bid, or a request for proposal can be submitted anywhere into the overlay network (e.g., by a third party insurance broker or an online client) and be routed toward a currently online, specific expert employee, speaking the same language as the requester. Note, the latter constraints are expressed as XPE filter expressions against which the XML document is evaluated in transit. This design fully decouples information requesters and information providers, avoids a single point of control and a single point of failure, and increases scalability due to decentralization and distribution.

This paper addresses the XML/XPath routing problem. More specifically, this paper focuses on the problem of efficiently routing an XML document emitted from a data producer at one point in the network to a set of data consumers

located anywhere throughout the network. Prior to receiving XML documents, consumers must have expressed interest in receiving XML documents by registering XPEs with the network. This problem statement is akin to the well-known publish/subscribe matching problem. However, the main difference here is that in the case of data dissemination networks there exists no one single centralized publish/subscribe system, but a network of content-based routers (i.e., a network or federation of publish/subscribe systems.) In the dissemination network, XML documents are routed based on their content and not based on IP address information, which is, due to the completely decoupled design, not available – all routing decisions are exclusively based on content information. Figure 1 provides an overview of the dissemination network this paper assumes. In the overlay network depicted in Figure 1 each broker only knows their neighbors (i.e., in terms of IP network address information.) However, none of the clients – neither data producers nor data consumers know about each other or about the network topology, except the router they connect to.

In the context of XML-based data dissemination, one of the main challenges is the ability to efficiently deliver relevant XML documents to a large and dynamically changing group of consumers. Centralized XML filtering [2, 8, 6] and distributed query-based XML retrieval approaches [9, 5, 15, 14] have found wide-spread interest, but do not address the distributed, content-based routing problem articulated above and addressed in this paper. Content-based routing [4, 19, 7] in distributed publish/subscribe architectures, have been studied for non-XML-based data. Their operational model assumes sets of attribute-value pairs joined by Boolean operators. It is not at all obvious how to extend these approaches for semi-structured data, especially due to the hierarchical data model of XML.

In this paper, we develop algorithms for dissemination of XML data throughout a network of content-based routers towards data consumers who have specified their interests through XPEs. Our contributions are: first, we adapt the use of advertisements to optimize data dissemination. While this idea is common in the publish/subscribe literature [4, 19, 7], it is not clear how to extend the concepts to the data model of XML. We demonstrate how to use the XML Document Type Definition (DTD) to generate advertisements about the information a data producer is going to publish. We distinguish between a non-recursive and a recursive case depending on the DTD defining the data emitting source. We then develop

*Middleware Systems Research Group, Technical Report, Oct 2006.

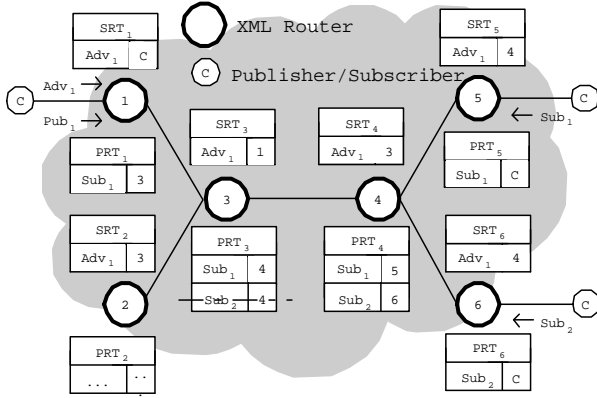


Figure 1. Content-based Routing

advertisement-based routing algorithms for both cases. Second, we propose a novel data structure to maintain XPEs by identifying the covering relations among them. We present covering algorithms for XPEs to reduce the routing table size stored at each router and speed up routing computation in the routers. Third, we present an optimization of merging similar XPEs to further reduce routing computation. Finally, we perform a detailed experimental evaluation of our approach on an overlay network comprised of 127 XML routers. Our experimental results demonstrate the effectiveness of the approach by reducing the routing table size by up to 90% and improving the routing time by roughly 85%.

2. Background

Content-based publish/subscribe systems [4, 19, 7] provide a flexible and extensible environment for information exchange. Messages in content-based pub/sub systems are routed based on their contents rather than the IP address of their destinations. In order to handle a large amount of dynamic information and reduce the network traffic, many optimization techniques, such as advertisements [4] and covering and merging techniques [7, 4, 19], have proven to gain significant benefits for non-XML based publish/subscribe systems. While conceptually, these ideas apply to XML-based data as well, it is not obvious how to apply these concepts to XML. This is the challenge addressed by this paper.

In advertisement-based publish/subscribe systems, advertisements are specifications of information that the publisher will publish in the future. Advertisements are flooded in the publish/subscribe overlay. The common assumption is that the number of advertisement is much less than that of subscriptions and publications. Advertisements are used to avoid broadcasting subscriptions in the network, so that subscriptions are only routed to the publishers who advertise what the subscribers are interested in. Subscriptions define filters on publications. Later on, only matching publications will be delivered to subscribers along the paths built by subscriptions.

Figure 1 shows a scenario for advertisement-based content-based routing. The subscription routing table (SRT) consisting of $\langle \text{advertisement, last-hop} \rangle$ tuples stores advertisements in order to route subscriptions. Publications will trace back along the path setup by subscriptions to interested subscribers. The publication routing table (PRT) maintains the path information. For example, in Figure 1, adv_1 is broadcast in the network, and is stored on each broker of the network with a different last hop. Consequently, subscriptions that match adv_1 will be routed according to these last hops (e.g., sub_1 is routed along the link $5 - 4 - 3 - 1$). Note that the sub_1 will not be forwarded to brokers 2 and 6 since the adv_1 indicates that the publisher is from broker 1. Therefore, the pub_1 is routed along a reverse path $1 - 3 - 4 - 5$ to the subscriber. In the rest of this paper, we use the notations $P(s)$ and $P(a)$ to refer to the set of publications that match subscription s and advertisement a , respectively.

The goal of covering-based routing [4, 19] is to remove redundant subscriptions from the network in order to maintain a compacted routing table and reduce the network traffic. In Figure 1, if sub_1 covers sub_2 , at broker 4, sub_2 will not be forwarded to broker 3. That is, we can safely remove sub_2 on broker 3 and gain a compacter routing table while maintaining the same information delivery behavior, for all the publications matching sub_2 must match sub_1 . Since advertisements have the same format as subscriptions, the covering relations among advertisements can be defined in the same way.

If two subscriptions do not have covering relations, but their publication sets overlap each other, the two subscriptions can be merged to a more general subscription, which covers the original subscriptions. Suppose sub_m is a merger of sub_1 and sub_2 , then we have $P(sub_m) \supseteq P(sub_1) \cup P(sub_2)$. There are two kinds of mergers. If the publication set of the merger is exactly equal to the union of the publication set of the original subscriptions, the merger is a *perfect merger*; otherwise, if $P(sub_m) \supset P(sub_1) \cup P(sub_2)$, it is an *imperfect merger*. After merging, only the merger is forwarded into the network. The merging technique [19] is used for further minimizing the routing table size. It is an extension of the covering technique.

3. Related Work

A large body of work has focused on developing publish/subscribe-style matching algorithms for evaluating an XML message against a set of XPEs [2, 8, 6, 3]. However, all these approaches exclusively address centralized matching architectures, not the distributed, content-based XML dissemination networks we address in this work. While matching is an integral step in a content-based router, other routing operations studied in this paper are equally important in a distributed data dissemination architecture. Thus, our work complements matching algorithms for the design of a content-based XML router.

Advertisement-based techniques for optimizing content-based routing have been developed in the area of distributed publish/subscribe [4, 19, 7]. It has been demonstrated that

the network traffic and routing table size can be reduced by using different routing strategies, including advertising, covering and merging techniques. However, the main differences between these approaches and our approach lie in the subscription language and publication data format. Our approach is based on the hierarchical, tree-based XPath and XML model; while the traditional content-based routing approaches operate with attribute-value pairs and predicate constraints over these pairs. The advertising carried out by XML and XPath data sources is different and more complex than predicate-based languages, for the hierarchical and recursive structure of the model needs to be taken into account. A DTD of an XML document does not have an equivalent in traditional publish/subscribe approaches. Galanis *et al.* [11] explore XML data dissemination based on a DHT. They use data summaries to ensure that queries are only sent to relevant servers. These data summaries could be taken as a form of advertisements. The data summary is generated from a XML document, so the expressiveness of the data summary is limited to part the DTD. Our contribution is to generate a complete advertisement set once from a DTD for all related XML documents.

The query aggregation scheme given in [5] addresses the problem of determining a compact set of XPEs from a given set of XPEs. This problem is similar to the covering and merging problem discussed in this paper. We think it could be used for the same purposes. However, it should guarantee the equivalence between the compact set and the original set. That is, the aggregate query set does not introduce false positives (i.e., takes XML documents not originally matched) or false negatives (i.e., misses XML documents originally matched.) These are non-trivial extensions to their work. Furthermore, the tree aggregation approach does not address the generation of advertisements from DTDs, which is central to our approach.

Recent research has focused on XML data dissemination [14, 9, 20, 15]. Koloniari *et al.* [14] present a decentralized approach for XML dissemination in a peer-to-peer network. However, in their approach queries are severely restricted in that no wildcards are allowed. Koudas *et al.* [15] propose a flexible routing protocol for XML routers to enable scalable XPath query and update processing in a data-sharing peer-to-peer network. Both approaches are solutions to the *location problem*. The location problem states that given a dynamic collection of XML database servers and an XPath query, find the databases that contain data relevant to the query. Our approach evaluates an XML document against a set of XPath queries, and decides where to route the XML document. Diao *et al.* [9] deploy XML-based services on an Internet-scale, and provides a detailed architectural design of the system. The NFA used in this work for matching naturally supports a form of specialized covering as XPEs with a common prefix share the NFA path taken while evaluating an input XML message against the XPEs indexed. In addition to prefix covering our approach identifies all cover relations among the XPEs processed and completely eliminates all covered XPEs from the routing computation. Moreover, our work introduces merg-

ing and advertising for XML data not addressed in the earlier approach. Theoretical properties of XPE containment are discussed in [17, 10]. They propose their own algorithms to detect the covering relations and give the computational complexity analysis for these algorithms, but none of them apply advertisement-based routing for XML dissemination. They do not consider XPE merging either.

4. Advertisement-based Routing

Upon receiving a subscription, a broker matches the subscription against its advertisements. If there is an advertisement whose publication set is overlapping with that of the subscription, it means there is a *match* between the subscription and the advertisement. The broker then routes the subscription to the broker where the advertisement came from. Before we discuss the matching algorithm for advertisement and subscription, we present the definition and format of advertisement for our approach.

4.1. Definition of Advertisement

In the context of XML/XPath routing, the advertisement is generated by exploiting DTD information. The purpose of a DTD is to define the legal building blocks of an XML document. The main building blocks of XML documents are elements surrounded with tags, e.g., $\langle root \rangle \dots \langle /root \rangle$. *root* is the element name in the documents. All elements appearing in the XML document must be defined in the corresponding DTD, which determines the structure of elements and their sequence in the document. In this paper, our discussion focuses on the main building block: elements. Our approach could be easily extended to element attributes and content, which we omit due to space limitations.

In this paper, we use the common interpretation of an XML document as a tree of nodes and consider each path from the root node to a leaf node. Thus, we decompose each XML document into a set of XML paths and each path is represented as $e = /t_1/t_2/\dots/t_n$, where t_i is the XML element name. These paths are extracted from the document before the publisher submits the document to the network. Thus, a publication routed in our system is actually an XML path annotated with a *pathId* and *docId*. This is transparent to publishers and subscribers who handle entire XML documents. Publishers submit entire XML documents, commonly referred to as publications, and subscribers submit XPath expressions (XPEs), commonly referred to as subscriptions. We use the terms XPE and subscription interchangeably in the rest of this paper.

We use the abstract XPath expression without *//*-operators as the format of advertisement in the context of XML/XPath data routing. Note that this is not a restriction of our subscription language. Advertisements are a system internal mechanism, which is not exposed to the application or the user. An advertisement is described as $a = /t_1/t_2/\dots/t_{n-1}/t_n$, where t_i can be either an element name or a wildcard, and a has the same length with the publication it advertises. In our approach,

advertisements are derived from the DTD, since the DTD contains all possible paths from the root to the leaves appearing in related XML documents.

We call an advertisement a *non-recursive* advertisement if it is extracted from a non-recursive DTD. Advertisement a is an example of non-recursive advertisement. A DTD is recursive if it has some element that is defined in terms of itself, directly or indirectly, e.g., the NITF DTD is recursive. We define an advertisement as a *recursive* advertisement if it has recursive elements defined in a DTD. An advertisement may have multiple recursive parts that appear in sequence or are embedded in each other. We classify the recursive advertisements to three categories as below.

Simple-recursive advertisements: There is only one recursive pattern in the simple-recursive advertisement. It is described as $a = /t_1/t_2/.../t_{i-1}(/t_i/.../t_j)^+/.../t_n$, where the $+$ operator declares that elements t_i, \dots, t_j must occur one or more times in the advertisement. Note that this is not part of XPath syntax, and advertisements are only used within the system, so the extended syntax has no effect to the clients and applications. In the proposed algorithms, we use $a = a_1(a_2)^+a_3$ to simplify the expression, where a_k ($1 \leq k \leq 3$) is a non-recursive advertisement.

Series-recursive advertisements: A series-recursive advertisement can include more than one recursive patterns in sequence. For example, the advertisement containing two recursive patterns in sequence can be described as $a = /t_1/t_2/.../t_{i-1}(/t_i/.../t_j)^+/t_{j+1}.../t_{l-1}(/t_l/.../t_o)^+/.../t_n$, or more simplified expression with non-recursive advertisements, $a = a_1(a_2)^+a_3(a_4)^+a_5$.

Embedded-recursive advertisements: In an embedded-recursive advertisement, recursive patterns can be embedded in others. A possible case is $a = /t_1/t_2/.../t_{i-1}(/t_i/.../t_{l-1}(/t_l/.../t_o)^+/.../t_j)^+/.../t_n$, or $a = a_1(a_2(a_3)^+a_4)^+a_5$. The embedded-recursive advertisement could be more complicated.

More types of recursive advertisements can be easily defined based on the above three types of advertisements. We discuss the matching algorithms for non-recursive and recursive advertisements in Sections 4.2 and 4.3, respectively.

4.2. Non-recursive Advertisement

In this section, we discuss the algorithms for subscription and non-recursive advertisement matching in the context of XML/XPath. An advertisement a matches a subscription s if the publication sets $P(a)$ and $P(s)$ overlap, that is, $P(a) \cap P(s) \neq \Phi$. Figure 2(a) shows all possible relations between two sets. To forward subscriptions, we need to identify the first two overlapping cases in Figure 2 (a). In this paper, we focus on the subscriptions including parent-child operator ($/$), wildcard operator ($*$), and ancestor-descendant operator ($//$). For other operators appearing in the subscription, such as attribute filters, our approach can be easily extended to support them through simple value comparisons. We discuss the matching algorithm in the following three subscription cases.

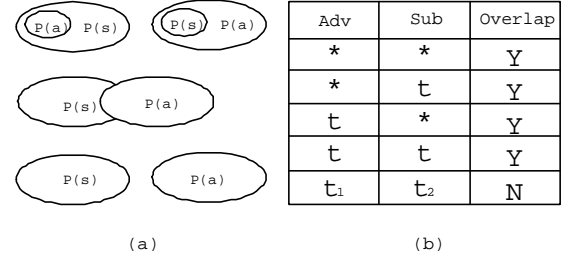


Figure 2. Adv. and Sub. Relations

Absolute simple XPEs: A simple XPE only contains parent-child and wildcard operators. Figure 3 shows the matching algorithm for absolute XPE (without $//$ -operator), e.g., $s = /st_1/st_2/.../st_k$, and advertisement, e.g., $a = /at_1/at_2/.../at_n$, where st_i is the i -th element for s and at_j is the j -th element for a . We will use these notations in all algorithms in this paper. The algorithm does not have to be applied, if the given XPE is longer than the advertisement (line 1). This observation is exploited because the advertisement has the same length as its publications, and thus, publications in $P(a)$ will not match all the elements in the longer XPE. Next, the algorithm compares each pair of elements or wildcards in advertisement and subscription (lines 2-3), according to the matching rules shown in Figure 2(b). It returns 0, if some pair does not overlap; otherwise it returns 1. For example, given $a = /b/*/* /c/c/d$ and $s = /* /c/* /b/c$, the algorithm return 0, since the matching rules fail to satisfy for $i = 4$. As shown in Figure 2(b), the fifth row indicates that the advertisement including an element c and the subscription including an element b at the same position do not overlap. That is publications matching the advertisement cannot match the subscription.

Input: advertisement a , and subscription s

Output: 1 if $P(a) \cap P(s) \neq \Phi$, 0 if $P(a) \cap P(s) = \Phi$

1: **If** $|s| > |a|$ **then return** 0

2: **For** $i = 1 : |s|$ **do**

3: **If** matching rules are not satisfied for at_i and st_i **then return** 0

4: **Return** 1

Figure 3. AbsExpr. and Adv. Matching

Relative simple XPEs: These expressions are similar to absolute simple XPEs except for the first operator, that is the XPE is relative. The matching could start at any position of the advertisement because the subscription is relative. A naive matching algorithm in this case is repeatedly calling Figure 3. In iteration i , take the subscription as an absolute one and start the matching from the i th bit of the advertisement. We skip the detail of the naive algorithm because it is straightforward. The complexity of the naive algorithm is $O(n * k)$ where n is the length of the advertisement and k is the length of the subscription. We propose an optimized version of the matching algorithm for relative simple XPEs.

The matching of relative simple XPE and advertisement is

a string matching problem [13]. We try to match the XPE, s , inside the advertisement, a , by starting at the first element of a that matches st_1 and continuing (i.e., comparing to st_2 and so on) until we either complete the match or find a mismatch. In the latter case, we must go back to the place from which we started. The difference between the traditional string matching problem and ours is that the $*$ can match any element in our matching rules, as shown in Figure 2(b). To improve this algorithm, the KMP algorithm [13] is applied to reduce the number of comparisons to $O(n)$. As shown in Figure 4, KMP computes a *next* table, recording all repeating patterns of s , to avoid backtracking. Figure 5 preforms the matching by taking advantage of the *next* table.

Input: subscription s
Output: *next* table for s (an array of size $|s|$)
1: $next(1) \leftarrow -1, next(2) \leftarrow 0$
2: **For** $i = 3 : |s|$ **do**
3: $j \leftarrow next(i - 1) + 1$
4: **While** matching rules are not satisfied for st_{i-1} and st_j ,
 and $j > 0$ **do** $j \leftarrow next(j) + 1$
5: $next(i) \leftarrow j$
6: **Return** *next*

Figure 4. Next Table for Sub.

Input: advertisement a , and subscription s
Output: *matchPos* if $P(a) \cap P(s) \neq \Phi$, 0 if $P(a) \cap P(s) = \Phi$
1: **If** $|s| > |a|$ **then return** 0
2: $j \leftarrow 1, i \leftarrow 1, matchPos \leftarrow 0$
3: **While** $matchPos = 0$ and $i \leq |a|$ **do**
4: **If** matching rules are satisfied for at_i and st_j **then** $j++ , i++$
5: **Else** $j \leftarrow next(j) + 1$, **if** $j = 0$ **then** $j \leftarrow 1, i++$
6: **If** $j = k + 1$ **then** $matchPos \leftarrow i - k$
7: **Return** *matchPos*

Figure 5. Optimized Matching

Descendant operators in XPEs: Descendant operators indicate that more than one element should appear in the matching advertisement. The matching of XPE with descendant operators and advertisement is based on the above XPE matching algorithms (i.e., Figure 3 and 5). We split a XPE into maximal length sub-XPEs that do not contain any descendant operators, and match each sub-XPE against the advertisement with sequence comparisons. We skip the detail of the algorithm and briefly describe it as below. The algorithm returns 1 if each sub-XPE matches different parts in a sequentially. First, the algorithm guarantees that the advertisement is longer than the subscription. Next, we match the first sub-XPE against the advertisement according to the different types of subscription, and recompute the next available matching position in the advertisement. The algorithm repeats this process until the end of the subscription is reached, or returns 0 immediately if it finds the rest of the advertisement is not long enough for the rest of sub-XPEs. For instance, given $a = /a/*e/*d/*c/b$ and $s = */a//d/*c//b$, the algorithm matches all sub-XPEs

in s against a in order. It returns 1 because it finds each sub-XPE $*/a, d/*c$ and b matches different parts, $a/*, */d/*$ and b , in a sequentially.

4.3. Recursive Advertisement

In this section, we mainly discuss the matching algorithms for XPE and recursive advertisement. According to the format of recursive advertisement defined in the Section 4.1, we discuss the matching algorithm for each type of recursive advertisement separately.

Input: advertisement $a = a_1(a_2)^+a_3$, and subscription s
($a_k, 1 \leq k \leq 3$, is an advertisement)
Output: 1 if $P(a) \cap P(s) \neq \Phi$, 0 if $P(a) \cap P(s) = \Phi$
01: **If** $|s| \leq |a_1a_2|$ **then return** AbsAdv(a_1a_2, s)
02: **Else** $temp \leftarrow$ AbsAdv($a_1a_2, /st_1/\dots/st_{|a_1a_2|}$)
03: **If** $temp = 0$ **then return** 0
04: **If** $|s| \leq |a_1a_2a_3|$ **then** $q \leftarrow 0$
05: **Else** $q \leftarrow Int((|s| - |a_1a_2a_3|)/|a_2|) + 1$
06: $p \leftarrow Int((|s| - |a_1a_2|)/|a_2|)$
07: **For** $c = q : p$ **do**
08: $temp \leftarrow$ AbsAdv($a_3, /st_{c*|a_2|+|a_1a_2|+1}/\dots/st_{|s|}$)
09: **If** $temp = 1$ **then return** 1
10: **If** $c = p$ **then** $temp \leftarrow$ AbsAdv($a_2, /st_{c*|a_2|+|a_1a_2|+1}/\dots/st_{|s|}$)
11: **Else** $temp \leftarrow$ AbsAdv($a_2, /st_{c*|a_2|+|a_1a_2|+1}/\dots/$
 $st_{(c+1)*|a_2|+|a_1a_2|}$)
12: **If** $temp = 0$ **then return** 0
13: **Return** 1

Figure 6. AbsExpr. & Simple RecAdv. Matching

We mainly focus on the matching for absolute XPE and recursive advertisements, for the matching of other types of XPEs and recursive advertisement can be implemented based on it. In Figure 6, the matching algorithm for absolute XPE and simple recursive advertisement calls the Figure 3 if the subscription is not longer than the end of the recursive pattern (line 1). If the subscription is longer, the algorithm estimates the maximum number that the recursive pattern would be repeated in the advertisement according to the length of both subscription and advertisement (lines 4-6). Next, the algorithm tries all possible advertisements according to the maximum number of repeated recursive pattern (lines 7-12). For example, given $a = /a/*c/(e/d)^+/*c/e$ and $s = /*c/e/d/*c/*d/e/d/*$, first, the algorithm compares $/a/*c/e/d$ in a with $/*c/e/d$ in s (line 2), and computes $q = 0$ and $p = 1$ from lines 4-6. Second, it supposes that the recursive pattern is repeated only once, compares $*/c/e$ in a with $e/d/*$ in s (line 8) and fails to match. Next, it repeats the recursive part e/d twice, and continues the comparisons (line 11). Finally, it returns 1 (line 9) when it finds a matches s with double recursive patterns in a . The complexity of the algorithm is $O(n^2)$, since it actually matches the subscription against each possible advertisement without recursive pattern. From a practical point of view, it would be reasonable to limit the maximum nesting depth of items in a document, which would reduce the complexity of processing DTDs.

Input: advertisement $a = a_1(a_2)^+a_3(a_4)^+a_5$, and subscription s
Output: 1 if $P(a) \cap P(s) \neq \Phi$, 0 if $P(a) \cap P(s) = \Phi$
1: **If** $|s| \leq |a_1a_2|$ **then return** $\text{AbsAdv}(a_1a_2, s)$
2: **Else** $p = \text{Int}((|s| - |a_1a_2|)/|a_2|) + 1$
3: **For** $c = 0 : p$ **do**
4: **return** $\text{SimRecAdv}(a_1(a_2)^{c+1}a_3(a_4)^+a_5, s)$

Figure 7. AbsExpr. & Series RecAdv. Matching

Input: advertisement $a = a_1(a_2(a_3)^+a_4)^+a_5$, and subscription s
Output: 1 if $P(a) \cap P(s) \neq \Phi$, 0 if $P(a) \cap P(s) = \Phi$
1: **If** $|s| \leq |a_1a_2a_3a_4|$ **then return** $\text{SimRecAdv}(a_1a_2(a_3)^+a_4, s)$
2: **Else** $p = \text{Int}((|s| - |a_1a_2a_3a_4|)/|a_2a_3a_4|) + 1$
3: **For** $c = 0 : p$ **do**
4: **return** $\text{SerRecAdv}(a_1(a_2(a_3)^+a_4)^{c+1}a_5, s)$

Figure 8. AbsExpr.& Embedded RecAdv.

Figure 7 describes the matching of absolute XPE and series-recursive advertisement. It is implemented by calling Figure 6 recursively. Figure 7 determines how many times the first recursive pattern could be repeated (line 2), and calls Figure 6 repeatedly (lines 3-4) to try all possible advertisement formats. $(a_2)^{c+1}$ in Figure 7 indicates that non-recursive advertisement a_2 will be repeated $c + 1$ times, thus, $a_1(a_2)^{c+1}a_3(a_4)^+a_5$ can be treated as a simple recursive advertisement. The matching of XPE and embedded recursive advertisement, shown in Figure 8, is similar to Figure 7. First, it determines how many times the outer recursive pattern could be repeated (line 2), and calls Figure 7 (not restricted to two recursive patterns) repeatedly (lines 3-4).

5. Covering and Merging

In this section, first, we describe a novel data structure called *subscription tree* for maintaining subscriptions. The data structure captures the covering relations among subscriptions. It can speed up the publication and subscription matching as well. In this paper we focus on advertisement-based subscription routing and its optimization techniques, such as covering and merging, the matching between XPE and XML data has been discussed in [12]. Second, we present the covering algorithms for absolute simple XPEs, relative simple XPEs and XPEs with descendant operator separately. Last, we explore the merging technique, and discuss the merging rules in the context of XPEs.

5.1 Subscription Tree

In covering-based routing, if an arriving subscription is covered by an existing subscription in the routing table, the new subscription is not forwarded to the next-hop broker. One the other hand, if the arriving subscription covers existing subscriptions, before it is forwarded, the broker needs to unsubscribe all the subscriptions that are covered by the new subscription. Therefore, the network traffic is reduced by removing the redundant subscriptions and the routing table in the

next-hop broker is compacted.

Subscriptions are maintained in a tree data structure. The idea is to store the subscriptions according to the covering relationship among them. A subscription at a node in the tree covers all subscriptions in its subtree. Since a covering relation defines a partial order among subscriptions, a tree data structure cannot capture all the covering relations. A subscription node can have only one parent in the tree, but it may be covered by several subscriptions. We allow each node having a set of *super pointers*, which indicate the covering relations with nodes outside its subtree, as shown in Fig 9. *Super pointers* are shortcuts to subscriptions that the node covers. With super pointers, a node covers its subtree, the nodes with subtrees pointed to by its super pointers, and the nodes with subtrees pointed to by its offsprings' super pointers.

The tree is maintained as follows. When a new subscription arrives, a breadth first search algorithm is used to search the tree in order to find a place to insert the subscription. At a given node the following three cases are distinguished.

Case 1: If the new subscription has no covering relationship with the node, the node's siblings are searched. If neither sibling has a covering relation with the new subscription, the subscription is inserted as new sibling. After insertion, super pointers maintained by the parent node are updated. If there is a super pointer of the parent pointing to a subscription that is also covered by the new subscription, then the super pointer is moved from the parent node to the new node.

Case 2: If the new subscription covers the current node, the new subscription is inserted between the current node and its parent. As a result, the new subscription becomes the parent of the current node, the old parent becomes the new subscription's parent. The old parent's super pointers are updated and moved to the new node, if there is a covering relation between the inserted subscription and a subscription pointed to by the super pointer.

Case 3: If the new subscription is covered by the current node, its children are searched until the new subscription is inserted. If the current node is a leaf node, the new node is inserted as the current node's child.

Existing super pointers are maintained while inserting. The new subscription may cause new covering relations and new super pointers are added. We can also add new super pointers

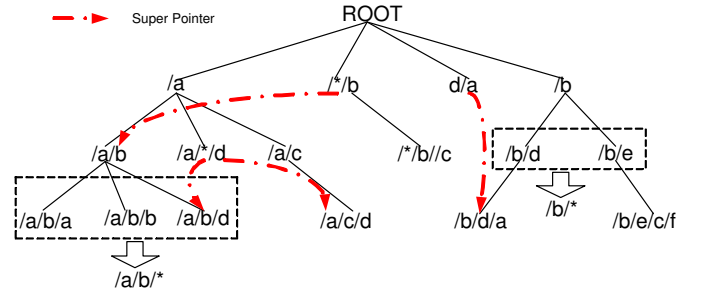


Figure 9. Subscription Tree

in proper nodes while searching the tree. However, this becomes expensive when the subscription tree grows larger. The reason we maintain the updated super pointers is for covering based routing. When a subscription arrives, if it is not covered by an existing subscriptions but it covers a set of subscriptions, we need to unsubscribe the subscriptions it covers and only forward the new subscription to neighbors. In this case, we need the super pointers to tell us what subscriptions should be unsubscribed. That means the updating of super pointers can be postponed until that point. With the existing pointers, we only search branches of the subscription tree that are not touched by existing super pointers.

Optimizations for the subscription searching and insertion can be performed based on the following two properties of the subscription tree.

Property of Absolute XPE node: For all the absolute simple XPEs which have no wildcard and //-operators, the children's path length is always longer than their parent's path length. The parent is the prefix of its children.

Based on this property, we can perform depth-first search for an XPE to find a start node which has the same length as itself and start breadth-first searching at that level. If an absolute XPE has a wildcard or //-operator in the middle of the expression, it is one or more levels higher than other simple XPEs of the same length in the subscription tree. Based on this property we can stop the search earlier.

Property of Relative XPE node: A relative XPE is a child node of either the root node or another relative node. It will never be inserted in a subtree rooted by an absolute XPE. This property reduces the search space in the subscription tree.

5.2. Covering Algorithm

The key problem is how to determine the relationship between two given subscriptions. The covering relation between subscriptions is the containment problem in the context of XPEs. It has been proven that containment of simple path expressions can be tested in PTIME [18]. It is studied as a part of the problem that checking/finding a prefix replacement for a simple query is in PTIME, in the context of semistructured database. In this section, we detect covering relations of XPEs containing wildcard, /- and //-operator in PTIME, and present covering rules and algorithms for determining covering relation between single path XPEs. The covering rules used in our algorithms are straightforward. We say Sub_1 containing an element t_i covers Sub_2 containing an element m_i at the corresponding position, if t_i is a wildcard no matter what m_i is, or $t_i = m_i$, where none of t_i and m_i is wildcard.

Absolute simple XPEs: The covering relation between two absolute XPEs (without //-operator) is the simplest case. Figure 10 shows the covering algorithm for two absolute XPEs. An important observation is that s_1 must be shorter than s_2 if s_1 wants to cover s_2 (line 1). It is exploited because a shorter XPE s has less constraints on items, and refers to a bigger matching set $P(s)$. Next, the algorithm compares each pair of $s_1 t_i$ and $s_2 t_i$ in s_1 and s_2 , respectively, according to the covering rules.

Input: two subscriptions s_1 and s_2

Output: 1 if s_1 covers s_2 , 0 if s_1 does not cover s_2

```

1: If  $|s_2| < |s_1|$  then return 0
2: For  $i = 1 : |s_1|$  do
3:   If covering rules are not satisfied for  $s_1 t_i$  and  $s_2 t_i$  then return 0
4: Return 1

```

Figure 10. Absolute Simple XPEs Covering

Input: two subscriptions s_1 and s_2 (s_1 is relative, s_2 is absolute or relative)

Output: $matchPos$ if s_1 covers s_2 , 0 if s_1 does not cover s_2

```

1: If  $|s_2| < |s_1|$  then return 0
2: For  $i = 1 : |s_2| - |s_1| + 1$  do
3:   If  $AbsCov(s_1, s_2 t_i / \dots / s_2 t_{|s_2|}) = 1$  then return  $matchPos = i$ 
4: Return 0

```

Figure 11. Relative Simple XPEs Covering

Relative simple XPEs: Figure 11 shows the covering algorithm for relative simple XPEs. It calls Figure 10 repeatedly (lines 2-3) to find if s_1 contains subscription s_2 or not. An absolute XPE s_1 can not cover a relative XPE s_2 , for the absolute XPE definitely refers to a smaller matching set $P(s_1)$ than $P(s_2)$.

It is important to note that the covering algorithm for relative simple XPEs is also a string matching problem, as we pointed out in Figure 5. The covering algorithm uses covering rules that are from subscription and advertisement matching rules used in Figure 5, however, a similar optimization can be applied to reduce the complexity of the covering algorithm from $O(k * n)$ to $O(k)$.

Descendant operators in XPEs: Figure 12 shows the covering algorithm for XPEs with descendant operators. It splits the XPE into sub-XPEs without //-operator, and matches each sub-XPE in s_1 against sub-XPEs in s_2 with sequence comparisons. Line 1 guarantees that s_2 is longer than s_1 . Next, it matches the first sub-XPE in s_1 against s_2 according to different types of s_1 and s_2 (lines 2, 5, 7). The algorithm moves to the next sub-XPE in s_1 if it finds a match, and moves to the next sub-XPE in s_2 if it does not find a match (lines 10-11 and 15-16). $temp$ is used to compute the next available matching position in s_2 . Finally, it returns 1 or 0 when it reaches the end of s_1 or s_2 , respectively (lines 19 and 14). For example, given $s_1 = /*/a//*/c$ and $s_2 = /a/a/*//c/e/c/d$, first, the algorithm compares the sub-XPE $*/a$ in s_1 with the sub-XPE $/a/a/*$ in s_2 (line 2). Second, it moves to the next sub-XPE $*/c$ in s_1 and compares it with $*$ in s_2 (lines 15-16). Next, it compares $*/c$ in s_1 with the next sub-XPE $c/e/c/d$ in s_2 (lines 10-11), and finally, it returns 1 from line 19 since the end of s_1 is reached and a match is found. Generally speaking, a sub-XPE in s_1 could not match a part of s_2 that includes a //-operator. For instance, given $s_1 = /*/a//*/c$ and $s_2 = /a/a/*//c/b/d$, the sub-XPE $*/c$ in s_1 does not cover $*/c$ in s_2 since $*/c$ refers to a smaller matching set. However, there is a special case that the sub-XPE s_{1i} in s_1 could cover a part of s_2 that includes a //-operator if s_{1i} ended with a wildcard and the matched part in s_2 ended with $//t$, where t

Input: two subscriptions $s_1 = s_{11}/\dots/s_{1q}$, and $s_2 = s_{21}/\dots/s_{2p}$
(both s_1 and s_2 can be relative or absolute subscription)

Output: 1 if s_1 covers s_2 , 0 if s_1 does not cover s_2

```

01: If  $|s_1| > |s_2|$  then return 0
02: If both  $s_1$  and  $s_2$  are absolute then
     $temp \leftarrow AbsCov(s_{11}, s_{21}), flag \leftarrow 0$ 
03: If  $temp = 0$ , and  $|s_{11}| = |s_{21}|$ , and  $s_{11}$  ends with *, and  $p \geq 2$ 
    then  $temp \leftarrow AbsCov(s_{11}t_1/\dots/s_{11}t_{|s_{11}|-1}, s_{21}), flag \leftarrow 1$ 
04: If  $temp = 0$  then return 0
05: Else if  $s_1$  is relative then  $temp \leftarrow RelCov(s_{11}, s_{21}), flag \leftarrow 0$ 
06: If  $temp = 0$ , and  $|s_{21}| \geq |s_{11}|$ , and  $s_{11}$  ends with *, and  $p \geq 2$ 
    then  $temp \leftarrow RelCov(s_{11}t_1/\dots/s_{11}t_{|s_{11}|-1},$ 
         $s_{21}t_{|s_{21}|-|s_{11}|+2}/\dots/s_{21}t_{|s_{21}|}), flag \leftarrow 1$ 
07: Else return 0
08:  $i \leftarrow 1, j \leftarrow 1, temp_m \leftarrow 0, temp_1 \leftarrow 0$ 
09: While  $i \leq q$  and  $j \leq p$  do
10: If  $temp = 0$  then  $j \leftarrow j + 1, temp_m \leftarrow 0$ 
11: If  $j \neq p + 1$  then
     $temp \leftarrow RelCov(s_{1i}, s_{2j}t_1+\dots/s_{2j}t_{|s_{2j}|}), flag \leftarrow 0$ 
12: If  $temp = 0$ , and  $|s_{2j}| \geq |s_{1i}| + temp_1$ , and  $s_{1i}$  ends with *,
    and  $p \geq j$  then  $temp \leftarrow RelCov(s_{1i}t_1/\dots/s_{1i}t_{|s_{1i}|-1},$ 
         $s_{2j}t_{|s_{2j}|-|s_{1i}|+2}/\dots/s_{2j}t_{|s_{2j}|}), flag \leftarrow 1$ 
13:  $temp_1 \leftarrow 0$ 
14: Else return 0
15:  $temp_m \leftarrow temp + temp_m + |s_{1i}|-1, i \leftarrow i + 1$ 
16: If  $i \neq q + 1$  and  $flag=0$  then
     $temp \leftarrow RelCov(s_{1i}, s_{2j}t_1+\dots/s_{2j}t_{|s_{2j}|}), flag \leftarrow 0$ 
17: If  $temp = 0$ , and  $|s_{2j}| \geq |s_{1i}| + temp_m$ , and  $s_{1i}$  ends with *,
    and  $p \geq j$  then  $temp \leftarrow RelCov(s_{1i}t_1/\dots/s_{1i}t_{|s_{1i}|-1},$ 
         $s_{2j}t_{|s_{2j}|-|s_{1i}|+2}/\dots/s_{2j}t_{|s_{2j}|}), flag \leftarrow 1$ 
18: Else if  $i \neq q + 1$  and  $flag=1$  then  $temp \leftarrow 0, temp_1 \leftarrow 1$ 
19: Else return 1

```

Figure 12. Descendant XPEs Covering

can be either a wildcard or an element (lines 3, 6, 12 and 17). For example, given $s_1 = /a/*//*/d$ and $s_2 = /a//b/c/d$, first, the algorithm compares $/a/*$ in s_1 with $/a//b$ in s_2 (line 3), where $flag = 1$ is used to record the current sub-XPE in s_1 matches a part of s_2 with $//$ -operator. Second, it moves to the next sub-XPE $*/d$ in s_1 (lines 15 and 18) and compares it with c/d in s_2 (line 11). Finally, it returns 1 since a match is found (line 19).

It is important to note that the covering detection between non-recursive advertisements is the same with the covering detection for subscriptions, since the non-recursive advertisement has the same format with an absolute simple subscription.

5.3. Merging

If there is no covering relation among a set of subscriptions, subscriptions can be merged into a new subscription to create a more concise routing table. In this section, we exploit the merging rules for XPEs.

In the subscription tree, child nodes of the same parent have a better chance to be merged. As shown in Fig 9, node $/a/b/a$, $/a/b/b$ and $/a/b/d$ can be merged and they are represented by a new node $/a/b/*$ which is a union of the original XPEs.

There was a super pointer pointing to node $/a/b/d$ before merging. This pointer should be removed because there is no covering relations between the pointer owner and the merger. If two nodes are merged, their subtrees become siblings of the merger. For example in Fig 9, after $/b/d$ and $/b/e$ are merged to $/b/*$, their children are the new node's children. The super pointer at $/b/d/a$ was not changed. To perform the merging in the subscription tree, we define several merging rules.

The subscriptions can be merged if they have only one difference (e.g., different elements). For instance, two subscriptions $s_1 = a/*/c/d$ and $s_2 = a/*/c/e$ can be merged into $s = a/*/c/*$. Note that if they differ in one operator, they should be in a covering relation to each other. The general form of this rule is:

- $s_1 = o_1 t_1 \dots o_i t_i o_{i+1} m o_{i+2} t_{i+1} \dots o_{n+1} t_n$
- $s_2 = o_1 t_1 \dots o_i t_i o_{i+1} k o_{i+2} t_{i+1} \dots o_{n+1} t_n$

are merged to

- $s = o_1 t_1 \dots o_i t_i o_{i+1} * o_{i+2} t_{i+1} \dots o_{n+1} t_n$

where m and k are different elements, o_i is either a $/$ -operator or a $//$ -operator, and t_i is a wildcard or an element.

Another rule is to merge subscriptions with two differences (e.g., different operators or different elements). For example, two subscriptions $s_1 = /a/c/*/*$ and $s_2 = /a//c/*/*$ that do not cover each other can be merged to $s = /a//c/*/*$. That is, different operators are merged to $//$ -operator, and different elements are merged to $*$. We represent the general form of this rule as:

- $s_1 = o_1 t_1 \dots o_i t_i o_{i+1} m \dots o_{j+1} t_j // t_{j+1} \dots o_{n+2} t_n$
- $s_2 = o_1 t_1 \dots o_i t_i o_{i+1} k \dots o_{j+1} t_j // t_{j+1} \dots o_{n+2} t_n$

are merged to

- $s = o_1 t_1 \dots o_i t_i o_{i+1} * \dots o_{j+1} t_j // t_{j+1} \dots o_{n+2} t_n$

where m , k and t_i is a wildcard or an element, and o_i is a $/$ -operator or a $//$ -operator.

A more general rule is to replace the different parts in two subscriptions with the $//$ -operator. We generalize this rule to:

- $s_1 = o_1 t_1 \dots o_i t_i XPE_1 o_{i+1} t_{i+1} \dots o_n t_n$
- $s_2 = o_1 t_1 \dots o_i t_i XPE_2 o_{i+1} t_{i+1} \dots o_n t_n$

are merged to

- $s = o_1 t_1 \dots o_i t_i // t_{i+1} \dots o_n t_n$

where XPE_1 and XPE_2 are different XPath expressions, t_i is a wildcard or an element, and o_i is a $/$ -operator or a $//$ -operator. This rule is applied if most parts in two subscriptions are equal, otherwise, more false positives will be introduced.

We periodically apply the above merging rules on the subscription tree to aggregate nodes that could be merged. We can compute an *imperfect merging degree* [16] if each broker in the network knows the DTD relative to the XML data producer. The imperfect degree of a new merger s , derived from s_1, s_2, \dots, s_n , is:

$$D_{imperfect} = \frac{|P(s) - \cup_{i=1}^n P(s_i)|}{|P(s)|}$$

It measures the imperfectness of an individual new merger. If the publications are distributed uniformly, the bigger the imperfect degree, the more false positive are introduced by the

new merger. For example, two subscriptions $s_1 = /a/* /c/d$ and $s_2 = /a/* /c/e$ can be merged into $s = /a/* /c/*$. If the corresponding DTD indicates that the elements a, b, c, d, e are allowed at the fourth position, 60% false positive will be introduced at position 4. We need to consider the distribution of other elements in the subscription, e.g., the probability of each element appearing at other positions, to compute the total number of false positive introduced. The false positives are not delivered to subscribers. It only happens in the network because of the imperfect merging. Clients are not exposed to the false positives.

6. Evaluation

In this section, we experimentally evaluate the performance of our routing and covering algorithms. All algorithms are implemented in C++. We perform all experiments on a computer with an Intel Xeon 2.4GHz processor with 2GB RAM. For generating the XPE workload, we use the XPath generator released by Diao *et al.* [8]. Queries are distinct, and we set the maximum length of an XPE to 10. We use the IBM XML Generator [1] to create the XML document workload. We use default parameters in this generator except that we set the maximum number of levels of the resulting XML documents to 10, which is consistent with the maximum length of XPEs. We use two different DTDs: the NITF (News Industry Text Format) DTD and the PSD (Protein Sequence Database) DTD. The performance metrics we measure include routing table size and publication routing time in a single broker, and the network traffic in broker topologies with 7 brokers and 127 brokers separately.

Routing Table Size: Our algorithms exploit the covering relations among XPEs. To verify this fact, we generate two data sets for NITF which include 100,000 XPEs each. We vary the probability of “*” occurring at a location step (W) and the probability of “//” occurring at a location step (DO) to generate two data sets A and B with different covering rates 90% and 50%, respectively. For each data set, we evaluate the effect of the covering optimization on routing table size. The routing table size is the number of XPEs in the table. As shown in Figure 13, for Set A , the routing table size is reduced dramatically by covering. The subscriptions in Set A have a higher degree of overlap. The results suggest that the covering algorithm perform better on data sets with higher degree of overlap. That is, covering technique gains more benefit when subscribers in a community has similar interests.

Publication Routing Time: In this experiment, we compare the covering-based routing time of each message with the routing time in the original system, using the above two data sets A and B . We generate 500 XML documents and extract 23,098 publications from these documents. Table 1 shows the routing time of the publications against 100,000 XPEs. The measurements are obtained by averaging the time taken to route all publications. Both Set A and Set B exhibit benefits, derived from subscription covering. After applying the covering algorithm, the routing time for Set A and Set B are

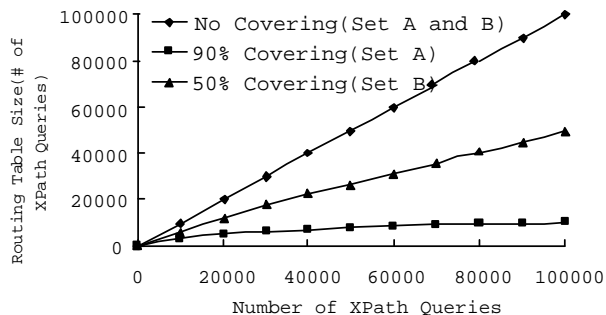


Figure 13. Routing Table Size

reduced by 84.6% and 47.5%, respectively.

Method	Set A (ms)	Set B (ms)
No Covering	13.96	14.23
Covering	2.15	7.47

Table 1. Publication Routing Performance

Network Traffic: The network traffic can be influenced by the broker topology, the distribution of subscribers and publishers, and the routing strategy. In this experiment, we investigate the impact of advertisement-based routing and covering techniques on network traffic, given a tree-like broker topology. As shown in Figure 14, the broker overlay network is a tree in which each broker is connected to 2 subordinate brokers. We build two overlays for the simulation. One has three levels, which consists of 7 brokers. The other broker overlay has seven levels with there are 64 leaf brokers, and 127 brokers in total. Each leaf broker is connected with a subscriber. We extend the size of the broker network to show the scalability of our approach. Publishers randomly connect to the broker overlays. In this experiment, we compare four routing

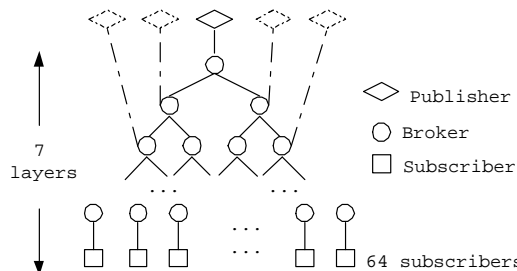


Figure 14. Broker Topology

strategies with different optimization techniques, including the routing with neither advertisement nor covering technique (no-Adv-no-Cov), the routing with covering only (no-Adv-with-Cov), the routing with advertisement only (with-Adv-no-Cov), and the routing with both advertisement and covering techniques (with-Adv-with-Cov). We generate 1,000 distinct XPEs for each sub-

Method	Network Traffic with 7 brokers	Network Traffic with 127 brokers
no-Adv-no-Cov	58,138	654,871
no-Adv-with-Cov	50,931	572,890
with-Adv-no-Cov	39,849	398,810
with-Adv-with-Cov	38,492	326,796

Table 2. Network Traffic (total number of messages received by all brokers in the network)

scriber using the PSD DTD, and 50 XML documents for the publishers. 4,182 publications are extracted from these documents. Table 2 shows the total number of messages in two broker overlays generated by one publisher. These messages, including advertisements, publications and subscriptions, are received by all brokers in the network under different routing strategies. As can be seen, the two advertisement-based routing methods significantly reduce the network traffic, because in this case a subscription is not flooded, and it is only forwarded to brokers that are on a path from the respective subscriber to the publisher. The introduction of advertisements reduces the network traffic to 68.5% and 75.6% for non-covering-based and covering-based routing strategies, respectively. Moreover, applying both advertisement-based routing and covering-based routing technique can reduce the overall network traffic to 66.2% and 49.9% in the two topologies. The experiment suggests that using advertisement to avoid subscription flooding and removing redundant queries by exploring covering relations among subscriptions can reduce the network traffic and save system resources. We gain more benefit in a larger broker network. The scalability of the system has been improved.

7. Conclusion

In this paper, we have studied the problem of efficiently routing XML data through a data dissemination network comprised of an overlay network of content-based routers. In the dissemination network, publishers' DTD files are transformed into advertisements expressed using XPath-like expressions. An advertisement creates a spanning tree rooted at the publisher. Subscribers specify their XPath filters which are forwarded along the reverse paths of *intersecting* advertisements, i.e., those may have potentially interesting XML documents. Now XML documents from publishers are forwarded along the reverse paths of *matching* XPEs to interested subscribers. In this paper, our contributions are: first, we discuss the advertisement-based routing protocol for XML-based data dissemination network. Advertisements are used to avoid the flooding of XPath queries, and can reduce the overall network traffic up to 75.6% according to the experiment. Advertisements are generated from a DTD file and can be applied for all related XML documents. We discuss both recursive and non-recursive advertisements extracted from DTD files. Moreover, we propose a set of algorithms for the matching of advertise-

ment and XPE. Second, we present some algorithms to determine the covering relations between to arbitrary XPEs, and a novel data structure called subscription tree to maintain the XPEs in an XML router, which maintains the covering relationship among XPEs. The routing time at each broker are improved by up to 85% in the most favorable cases because of a compact routing table, which is generated by removing redundant XPEs after exploring the covering relationship. Third, we also propose some rules to merge similar XPEs in order to further reduce the routing table size. The evaluation results suggest that the scalability of the system has been improved by applying advertisement-based routing, covering, and merging techniques.

References

- [1] A.L.Diaz and D.Lovell. XML generator, Sept. 2003.
- [2] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *VLDB*, 2000.
- [3] N. Bruno, L. Gravano, and N. Doudas. Navigation-vs. index-based XML multi-query processing. In *ICDE*, 2003.
- [4] A. Carzaniga, M. J. Rutherford, and A. L. Wolf. A routing scheme for content-based networking. In *IEEE INFOCOM*, 2004.
- [5] C. Chan, W. Fan, P. Felber, and M. Garofalakis. Tree pattern aggregation for scalable XML data dissemination. In *VLDB*, 2002.
- [6] C. Chan, P. Felber, and M. Garofalakis. Efficient filtering of XML documents with XPath expressions. In *ICDE*, 2002.
- [7] G. Cugola, E. D. Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE TSE*, 2001.
- [8] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *ACM Trans. Database Syst.*, 2003.
- [9] Y. Diao, S. Rizvi, and M. Franklin. Towards an internet-scale XML dissemination service. In *VLDB*, 2004.
- [10] X. Dong, A. Halevy, and I. Tatarinov. Containment of nested XML queries. In *TR UW-CSE-03-12-05, U. of Washington*, 2003.
- [11] L. Galanis, Y. Wang, S. Je, and E. DeWitt. Locating data sources in large distributed systems. In *VLDB*, 2003.
- [12] S. Hou and H.-A. Jacobsen. Predicate-based filtering of XPath expressions. In *ICDE*, 2006.
- [13] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 1977.
- [14] G. Koloniari and E. Pitoura. Content-based routing of path queries in peer-to-peer systems. *EDBT*, 2004.
- [15] N. Koudas, M. Rabinovich, and D. Srivastava. Routing XML queries. *ICDE*, 2004.
- [16] G. Li, S. Hou, and H.-A. Jacobsen. A unified approach to routing, covering and merging in publish/subscribe systems based on modified binary decision diagrams. *ICDCS*, 2005.
- [17] G. Miklau and D. Suciu. Containment and equivalence for a fragment of xpath. *J. ACM*, 2004.
- [18] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT*, 1999.
- [19] G. Mühl. Large-scale content-based publish/subscribe systems. *Ph.D Dissertation*, University of Darmstadt, 2002.
- [20] A. C. Snoeren, K. Conley, and D. K. Gifford. Mesh-based content routing using XML. *SIGOPS*, 2001.