Efficient Subscription Management in Content-based Networks

Raphaël Chand, Pascal A. Felber
Institut EURECOM
06904 Sophia Antipolis, France
{chand|felber}@eurecom.fr

Abstract

Content-based publish/subscribe systems offer a convenient abstraction for data producer and consumers, as most of the complexity related to addressing and routing is encapsulated within the network infrastructure. A major challenge of content-based networks is their ability to efficiently cope with changes in consumer membership. In our XNET *XML content network, we have addressed this issue by designing novel algorithms to speed up subscription management at the routers, while guaranteeing perfect routing at all times and maintaining compact routing tables thanks to extensive usage of "aggregation." In this paper, we discuss the issue of subscription management in content-based networks, and we specifically describe and evaluate the algorithms that we have developed for* XNET.

1 Introduction

In content-based publish/subscribe systems, messages are routed on the basis of their content and the interests (subscriptions) of the message consumers. This form of communication is well adapted to loosely-coupled distributed systems with large consumer populations, with diverse interests, wide geographical dispersion, and heterogeneous resources (e.g., CPU, bandwidth). Several techniques have been proposed to implement content routing, with various trade offs in terms of algorithmic complexity, runtime overhead, or bandwidth utilization. In particular, support for *perfect* routing (i.e., a message traverses a communication link only if there is some consumer downstream that is interested in that message) introduces significant management complexity at the routers in the presence of dynamic subscription registrations and cancellations. As a consequence, several content-based publish/subscribe networks do not systematically update their routing tables upon consumer departure and let the accuracy of routing degrade over time.

In our XNET XML content network [7], we have addressed this issue by designing novel techniques to speed up the most time-consuming subscription management operation of the routers. Specifically, we propose algorithms that allow routers to quickly determine the "covering" relationships between an incoming subscription and all the entries of their routing table. Covering relationships are at the core of *subscription aggregation* mechanisms, which help limit the size of routing tables and hence improve the efficiency of the filtering engine while ensuring perfect routing. Although the algorithms presented in this paper have been designed for XPath tree-structured subscriptions, they can be readily applied to other subscription language with similar characteristics. Experimental evaluation demonstrates that these algorithms are highly efficient even when the number of subscriptions in the routing table grows very large.

2 Related Work

Several publish/subscribe systems implement some form of distributed content based routing, most notably IBM Gryphon [2], Siena [3], and Jedi [10]. These systems adopt various approaches to subscription management.

IBM Gryphon [2] uses a distributed filtering algorithm based on parallel search trees maintained on each of the brokers to efficiently determine where to route the messages. The system implements perfect routing and supports subscription registration and cancellations; in fact, registering (canceling) a subscription reduces to inserting (removing) it from the search tree and is thus an efficient operation. However, to maintain and update the parallel search tree, each broker must have a copy of all the subscriptions in the system. As a consequence, this approach may not scale well to large and highly dynamic consumer populations.

Siena [3] also uses a network of event servers for content-based event distribution, and relies upon a routing protocol most similar to ours, but with limited support for subscription cancellation. In a recent paper [4], the authors of Siena introduce a novel routing scheme for content-based networking based on a combination of broadcast and selective routing. Subscription management is simple and efficient but the system does not guarantee perfect routing, in the sense that consumers may receive messages that they are not interested in. The authors have addressed this issue by having routers periodically request for the routing table of other routers.

JEDI [10] proposes several variations for routing events among its networked event servers; in particular, with the *hierarchical* approach, subscriptions are propagated upward a spanning tree and messages are propagated both upward and downward to the children that have matching subscriptions. Subscription management is simple and efficient, but this approach may lead to large routing tables at the root and unnecessary propagation of events upward the tree.

Our subscription containment and matching techniques are related to the widely studied problem of pattern and regular expression matching. There exists several indexing methods to speed up the search of textual data with regular expressions, like the bit-parallel implementation of NFA [1] and suffix trees [14]. In [6], the authors have addressed the reverse indexing problem of retrieving all the regular expressions that match a given string. They propose RE-Tree, an index structure to quickly determine the regular expressions that match a given input string, by focusing the search on only a small fraction of the expressions in the database.

In [12], Tozawa and Hagiya present a containment checking technique for XML schemas, which is based on binary decision diagrams. Little work has been done on the problem of containment checking for tree-structured XPath expressions. In fact, the problem has been shown to be coNP-complete [11]. A sound but non-complete algorithm has been proposed in [5] to determine whether a given tree-structured subscription covers another subscription, but it does not address the problem of covering relationships between large sets of subscriptions.

3 System Overview

This section gives an overview of the XNET content routing network. We also briefly describe its most essential mechanisms, which are relevant for the rest of the paper. More details can be found in [7].

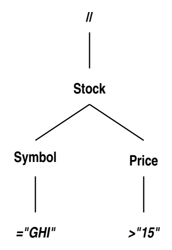
System Model and Definitions. XNET is a distributed publish/subscribe system which consists of a collection of content-based routers (or nodes) organized in an overlay network. Each node routes the messages based on its local knowledge of the consumer subscriptions and the actual content of the messages.

Each data consumer and producer is connected to some node in the network; we call such nodes *consumer* and *producer* nodes. We assume that all routers know their neighbors, as well as the best paths that lead to each producer. We also assume that the number and location of the producer nodes is known. From the point of view of a router, this amounts to knowing which neighbors lead to some producer. The consumer population can be highly dynamic and does not need to be known a priori. The most recent implementation of our routing protocol, XROUTE, handles multiple producers [8]; for the sake of simplicity, however, we only consider networks with a single producer in this paper.

Each node has a set of *links*, or *interfaces*, that connects the node to its direct neighbors. We assume that there exists exactly one interface per neighbor, and that communication between two nodes is reliable. Our system also incorporates fault-tolerant mechanisms to handle both transient and permanent failures.

```
<Quotes>
  <Stock>
    <Symbol>DEF</Symbol>
    <Price>34.1</Price>
  </Stock>
  <Stock>
    <Symbol>GHI</Symbol>
    <Price>11.5</Price>
  </Stock>
</Quotes>
```

(a)



(b)

Figure 1: (a) A sample XML document describing two stock quotes. (b) Tree representation of a sample XPath subscription (`//Stock[Symbol="GHI"][Price>15]`) that does not match the XML document.

XNET was designed to deal with XML data, the *de facto* interchange language on the Internet. Producers can thus define custom data types and generate arbitrary semi-structured events, as long as they are well-formed XML documents.

Consumer interests are expressed using a subscription language. Subscriptions allow to specify predicates on the set of valid events for a given consumer. XNET uses a significant subset of the standard XPath language to specify complex, tree-structured subscriptions [13].

An XPath expression contains one or more *location steps*, separated by slashes (`/`). In its most basic form, a location step designates an element name followed by zero or more predicates specified between brackets. Predicates generally specify constraints on the presence of structural elements, or on the values of XML documents using basic comparison operators. XPath also allows the use of wildcard (`*`) and ancestor/descendant (`//`) operators, which respectively match exactly one and an arbitrarily long sequence of element names. We say that an XML document *matches* an XPath expression when the evaluation of the expression on the document yields a non-null object. Figure 1 shows an XML event and an XPath subscription that does not match the event (each branch of the subscription has a matching node in the XML document, but the conjunctive condition at the "Stock" node is not met).

We say that a subscription S_1 *covers* or *contains* another subscription S_2, denoted by $S_1 \supseteq S_2$, iff any event matching S_2 also matches S_1, i.e., *matches*$(S_2) \Rightarrow$ *matches*(S_1). The covering relationship defines a partial order on the set of all subscriptions.

The Routing Protocol. XNET implements *perfect routing*, that is, a message traverses a communication link only if there is some consumer downstream that is interested in that message. To do so, each node in the network maintains in its routing table a collection of subscriptions that describe the classes of message that its neighboring nodes are interested in. When receiving a message, a node first determines which subscriptions of its routing table match the event; it then forwards the message to all neighbors that have registered one of these subscriptions. Given accurate routing tables, this process ensures that a message eventually reaches all the consumers, and only those, that are interested in that message.

When a consumer registers or cancels a subscription, the nodes of the overlay update their routing table accordingly by exchanging some pieces of information that represent the registration or cancellation of the consumer. The process starts at the consumer node and terminates at the producer node(s), following the shortest paths. As a consequence, messages published by the producers follow the reverse paths of the subscriptions, along a multicast tree spanning all interested consumers.

The routers in our system reduce the size of their routing tables as much as possible by using elaborate *aggregation* techniques, which are based on the detection and the elimination of subscription redundancies. Subscription aggregation allows us to dramatically improve the routing efficiency of the system both in terms of throughput and latency, because the time necessary to filter a message is proportional to the number of entries in the routing tables. On the other hand, aggregation also adds significant complexity and overhead to the routers, because they need to identify the covering relationships between incoming subscriptions and all the entries of their routing tables. These management operations were the main bottleneck of early XNET implementations and led us to develop the techniques presented hereafter. More details about subscription aggregation and XNET's routing protocol can be found in [7].

4 Subscription Management

Efficient subscription management is critical for the overall performance of the system and to guarantee short registration delays to consumers. As previously mentioned, the cost of subscription management mainly results from the extensive covering checks that have to be performed by the routers when a subscription is registered or canceled.

To determine whether a given tree-structured subscription—also called "tree pattern" henceforth—covers another subscription, we can use the algorithm proposed in [5], which has a time complexity of $O(|S_1||S_2|)$, where $|S_1|$ and $|S_2|$ are the number of nodes of the two subscriptions being compared.[1] Obviously, when an incoming subscription must be tested for covering against all the other subscriptions in the routing table, iterative execution of the algorithm is clearly inefficient. We have therefore designed a novel algorithm, termed XSEARCH, which efficiently identifies all the possible covering relationships between a given subscription and a possibly large set of subscriptions. This algorithm is described in the rest of this section. Additional details and proofs can be found in a companion technical report [8].

Problem Statement. Consider a tree pattern s and a set R of n tree patterns, $R = \{s_1, \cdots, s_n\}$, which we will refer to as the search set. Our algorithm runs in two different modes according to the relationships that we want to identify. *Covered mode* identifies the set $R_{\supseteq}$ of all the tree patterns in R that are covered by s. *Cover mode* identifies the set $R_{\subseteq}$ of all the tree patterns in R that cover s. We refer to XSEARCH$_{\supseteq}$ and XSEARCH$_{\subseteq}$ as the algorithm running in *covered* and *cover* mode, respectively.

Definitions and Notations. Let u be a node of a tree pattern s; we denote by *label*(u) the label of that node and by *child*(u) the set of the child nodes of u in s. Recall that the label of node u can either be a wildcard (`*`), an ancestor/descendant operator (`//`), or a tag name. We define a partial ordering $\preceq$ on node labels such that if x and x' are tag names, then (1) $x \preceq * \preceq //$ and (2) $x \preceq x'$ iff $x = x'$.

Algorithm 1 *add*(s, t, u)

```
1: if $\exists t' \in child(t)$ such that $label(t') = label(u)$ and $s \notin sub(t')$ then
2:     $sub(t') = sub(t') \cup s$
3: else
4:     create $t' \in child(t)$ such that $label(t') = label(u)$ and $sub(t') = \{s\}$
5: end if
6: for all $u' \in child(u)$ do
7:     $add(s, t', u')$
8: end for
```

Factorization Trees. Our XSEARCH algorithm does not operate directly on the set of tree patterns R, but on a "factorization tree" built from the set R and defined as follows. The factorization tree of R, denoted $T(R)$, is a tree where each node t has two attributes: a label *label*(t) similar to that of a node of a tree pattern, and a set of tree patterns *sub*(t), which is a subset of R. The root node r_T of $T(R)$ has no label and *sub*$(r_T) = R$. Initially, $T(R)$ consists of only its root node r_T. We incrementally add each tree pattern $s \in R$ to $T(R)$ with the recursive *add*(s, r_T, r_s) function shown in Algorithm 1, where r_s is the root node of tree pattern s. The removal of a tree pattern from $T(R)$ is performed in a similar manner using Algorithm 2. Note that, to keep the presentation simple, we omitted the special

[1]Note that the covering problem has been shown to be coNP-complete [11]. Our algorithm is sound but not complete, i.e., it may fail to detect some covering relationships in rare pathological cases, but all the relationships that it reports are correct. Consequently, a router may fail to aggregate some valid subscriptions, but correctness is never violated.

case of the root node of the factorization tree in the addition and removal algorithms.

Algorithm 2 *remove*(s, t)

```
1: for all $t' \in child(t)$ such that $s \in sub(t')$ do
2:    $sub(t') = sub(t') \setminus \{s\}$
3:    $remove(s, t')$
4:    if $sub(t') = \{\}$ then
5:       remove $t'$ from $child(t)$
6:    end if
7: end for
```

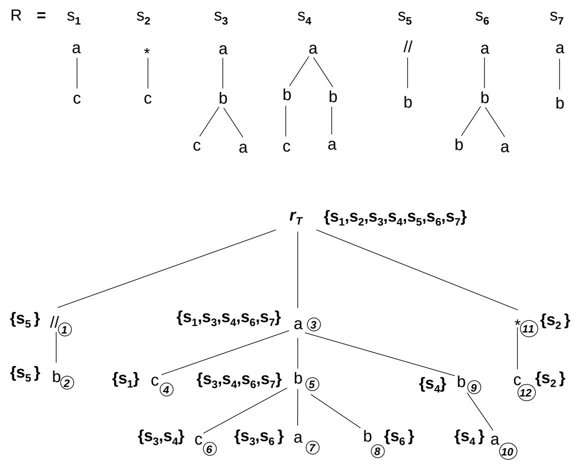


Figure 2: Six tree patterns and a corresponding factorization tree, where a node is represented by its label. Each node is associated with a set of tree patterns, shown between brackets.

Intuitively, a factorization tree enables us to remove the redundancies between the tree patterns in R by "factorizing" identical branches. Thus, $T(R)$ is a compact representation of the tree patterns in R. Figure 2 shows an example with six tree patterns and the corresponding factorization tree. It is important to note that the factorization tree is not unique; depending on the insertion order of the tree patterns, we can have distinct, equivalent trees. This does not affect the correctness of our XSEARCH algorithm, nor its performance.

The Search Algorithm. We first describe the XSEARCH algorithm in covered mode. Consider a subscription set R and a corresponding factorization tree, $T(R)$. Let s be a single tree pattern. The algorithm works recursively on the nodes of s. When executed with the root nodes of $T(R)$ and s, $\text{XSEARCH}_{\supseteq}(r_T, r_s)$ returns the set $R_{\supseteq}$ of all tree patterns that are covered by s.

The search process is described in pseudo-code in Algorithm 3. Intuitively, it tries to locate the paths in $T(R)$ that are covered by s; the tree patterns that the union of those paths represent are also covered by s (lines 6 and 8). The process is slightly more complex when encountering an ancestor/descendant operator (`//`), because we need to try to map it to paths of length 0 (line 11) or ≥ 1 (line 12).

To better illustrate the workings of the algorithm, consider the example runs shown in Figure 3. Two tree patterns, u and v, are matched against the factorization tree $T(R)$ of Figure 2 for clarity. The nodes of u, v, and $T(R)$ are numbered in the figures; we refer to node number i of u, v, and $T(R)$ by u_i, v_i, and t_i, respectively. The different steps of the algorithm are detailed in the two execution traces (the $\hookrightarrow$ symbol represents recursive invocations of the algorithm).

Algorithm 3 $\text{XSEARCH}_{\supseteq}(t, u)$

```
1: if $t$ is a leaf then
2:    $\text{XSEARCH}_{\supseteq}(t, u) = \emptyset$
3: else
4:    if $label(u) \neq$ "$//$" then
5:       if $u$ is a leaf then
6:          $\text{XSEARCH}_{\supseteq}(t, u) = \bigcup_{label(t') \preceq label(u)}^{t' \in child(t)} sub(t')$
7:       else
8:          $\text{XSEARCH}_{\supseteq}(t, u) = \bigcup_{label(t') \preceq label(u)}^{t' \in child(t)} \bigcap_{u' \in child(u)} \text{XSEARCH}_{\supseteq}(t', u')$
9:       end if
10:   else
11:      $S_0 = \bigcap_{u' \in child(u)} \text{XSEARCH}(t, u')$
12:      $S_{\geq 1} = \bigcup_{t' \in child(t)} \text{XSEARCH}(t', u)$
13:      $\text{XSEARCH}_{\supseteq}(t, u) = S_0 \cup S_{\geq 1}$
14:   end if
15: end if
```

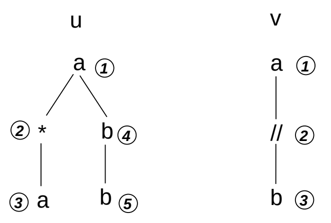
Algorithm 4 $\text{XSEARCH}_{\subseteq}(t, u)$

```
1: if $u$ is a leaf then
2:    $\text{XSEARCH}_{\subseteq}(t, u) = sub(t)$
3: else
4:    if $label(t) \neq$ "$//$" then
5:       if $\nexists u' \in child(u), label(u') \preceq label(t)$ then
6:          $\text{XSEARCH}_{\subseteq}(t, u) = sub(t)$
7:       else
8:          if $t$ is a leaf then
9:             $\text{XSEARCH}_{\subseteq}(t, u) = \emptyset$
10:         else
11:            $\text{XSEARCH}_{\subseteq}(t, u) =$
12:            $\bigcap_{label(u') \preceq label(t)}^{u' \in child(u)} \bigcup_{t' \in child(t)} \text{XSEARCH}_{\subseteq}(t', u')$
13:         end if
14:      end if
15:   else
16:      $S_0 = \bigcup_{t' \in child(t)} \text{XSEARCH}(t', u)$
17:      $S_{\geq 1} = \bigcap_{u' \in child(u)} \text{XSEARCH}(t, u')$
18:      $\text{XSEARCH}_{\subseteq}(t, u) = S_0 \cap S_{\geq 1}$
19:   end if
20: end if
```

Algorithm 5 $\text{XSEARCH}_{\subseteq}(r_T, r_s)$

```
1: $\text{XSEARCH}_{\subseteq}(r_T, r_s) = sub(t) \setminus \bigcup_{t' \in child(t)} \text{XSEARCH}_{\subseteq}(t', r_s)$
```

The second algorithm, $\text{XSEARCH}_{\subseteq}$, is described in Algorithms 4 and 5 and works in a very similar manner. The major difference is that the algorithm works recursively on the nodes of $T(R)$, trying to find paths in s that are covered by the tree patterns in $T(R)$. The recursive function in Algorithm 4 returns the subscriptions that do *not* cover s. A subscription t covers s if each branch of s is covered by some branch of t (line 12). Subscriptions that have longer (line 2) or incompatible (line 6) paths cannot cover s, whereas shorter paths (line 9) are acceptable. Finally, when encountering an ancestor/descendant operator (`//`), we need to try to map it to paths of length 0 (line 16) or



$\text{XSEARCH}(r_T, u_1) = \text{XSEARCH}(t_3, u_2) \cap \text{XSEARCH}(t_3, u_4)$
$\hookrightarrow \text{XSEARCH}(t_3, u_2) = \text{XSEARCH}(t_4, u_3) \cup \text{XSEARCH}(t_5, u_3) \cup \text{XSEARCH}(t_9, u_3)$
$\quad \hookrightarrow \text{XSEARCH}(t_4, u_3) = \emptyset$
$\quad \hookrightarrow \text{XSEARCH}(t_5, u_3) = \{s_3, s_6\}$
$\quad \hookrightarrow \text{XSEARCH}(t_9, u_3) = \{s_4\}$
$\hookrightarrow \text{XSEARCH}(t_3, u_2) = \{s_3, s_6\} \cup \{s_4\} = \{s_3, s_4, s_6\}$
$\hookrightarrow \text{XSEARCH}(t_3, u_4) = \text{XSEARCH}(t_5, u_5) \cup \text{XSEARCH}(t_9, u_5)$
$\quad \hookrightarrow \text{XSEARCH}(t_5, u_5) = \{s_6\}$
$\quad \hookrightarrow \text{XSEARCH}(t_9, u_5) = \emptyset$
$\hookrightarrow \text{XSEARCH}(t_3, u_4) = \{s_6\}$

Finally: $\text{XSEARCH}(r_T, u_1) = \{s_3, s_4, s_6\} \cap \{s_6\} = \{s_6\}$

$\text{XSEARCH}(r_T, v_1) = \text{XSEARCH}(t_3, v_2)$
$\hookrightarrow \text{XSEARCH}(t_3, v_2) = S_0 \cup S_{\geq 1}$
$\hookrightarrow S_0 = \text{XSEARCH}(t_3, v_3)$
$\hookrightarrow S_{\geq 1} = \text{XSEARCH}(t_4, v_2) \cup \text{XSEARCH}(t_5, v_2) \cup \text{XSEARCH}(t_9, v_2)$
$\quad \hookrightarrow \text{XSEARCH}(t_3, v_3) = \textit{sub}(t_5) \cup \textit{sub}(t_9) = \{s_3, s_4, s_6, s_7\}$
$\quad \hookrightarrow \text{XSEARCH}(t_4, v_2) = \emptyset$
$\quad \hookrightarrow \text{XSEARCH}(t_5, v_2) = S'_0 \cup S'_{\geq 1}$
$\qquad \hookrightarrow S'_0 = \text{XSEARCH}(t_5, v_3)$
$\qquad \hookrightarrow S'_{\geq 1} = \text{XSEARCH}(t_6, v_2) \cup \text{XSEARCH}(t_7, v_2) \cup \text{XSEARCH}(t_8, v_2)$
$\qquad\quad \hookrightarrow \text{XSEARCH}(t_5, v_3) = \textit{sub}(t_6) = \{s_6\}$
$\qquad\quad \hookrightarrow \text{XSEARCH}(t_6, v_2) = \emptyset$
$\qquad\quad \hookrightarrow \text{XSEARCH}(t_7, v_2) = \emptyset$
$\qquad\quad \hookrightarrow \text{XSEARCH}(t_8, v_2) = \emptyset$
$\qquad \hookrightarrow S'_0 = \{s_6\}$
$\qquad \hookrightarrow S'_{\geq 1} = \emptyset$
$\quad \hookrightarrow \text{XSEARCH}(t_5, v_2) = \{s_6\}$
$\quad \hookrightarrow \text{XSEARCH}(t_9, v_2) = S''_0 \cup S''_{\geq 1}$
$\qquad \hookrightarrow S''_0 = \text{XSEARCH}(t_9, v_3)$
$\qquad \hookrightarrow S''_{\geq 1} = \text{XSEARCH}(t_{10}, v_2)$
$\qquad\quad \hookrightarrow \text{XSEARCH}(t_9, v_3) = \emptyset$
$\qquad\quad \hookrightarrow \text{XSEARCH}(t_{10}, v_2) = \emptyset$
$\qquad \hookrightarrow S''_0 = \emptyset$
$\qquad \hookrightarrow S''_{\geq 1} = \emptyset$
$\quad \hookrightarrow \text{XSEARCH}(t_9, v_2) = \emptyset$
$\hookrightarrow S_0 = \{s_3, s_4, s_6, s_7\}$
$\hookrightarrow S_{\geq 1} = \{s_6\}$
$\hookrightarrow \text{XSEARCH}(t_3, v_2) = \{s_3, s_4, s_6, s_7\} \cup \{s_6\} = \{s_3, s_4, s_6, s_7\}$

Finally: $\text{XSEARCH}(r_T, v_1) = \{s_3, s_4, s_6, s_7\}$

Figure 3: Two XSEARCH$_\supseteq$ example runs.

≥ 1 (line 17). Note that we implicitely introduce an artificial root node in the tree-structured subscriptions (denoted r_s for subscription s) in order to simplify the description of the algorithm. When called with the roots of the factorization tree and a subscription s, Algorithm 5 recusively searches for subscriptions that do not cover s and return the complement set with respect to R. Because of space limitations, correctness proofs are not given here (they can be found in [8]).

Both Algorithms 3 and 4 perform in $O(|T(R)| \cdot |s|)$ time, where $|T(R)|$ is the number of nodes in the factorization tree and $|s|$ that in the expression being tested. This quadratic time complexity is due to the fact that each node in $T(R)$ and s is checked at most once. As for the space complexity, the size of the factorization tree $T(R)$ grows linearly with the number of tree patterns in the search set R. However, by construction, the factorization tree typically requires much less space than would be needed to maintain the whole search set R, that is, $|T(R)| \ll \sum_{s_i \in R} |s_i|$ when $|R|$ grows to large values.

5 Performance evaluation

Experimental Setup. To test the effectiveness of our subscription management techniques, we have conducted simulations using real-life document types and large numbers of subscriptions. We have generated realistic subscription workload using a custom XPath generator that takes a Document Type Descriptor (DTD) as input and creates a set of valid XPath expressions based on a set of parameters that control: the maximum height h of the tree patterns; the probabilities p_* and $p_{//}$ of having a wildcard (`*`) and ancestor/descendant (`//`) operators at a node of a tree pattern; the probability p_λ of having more than one child at a given node; and the skew θ of the Zipf distribution used for selecting element tag names. For our experiments, we generated sets of tree patterns of various sizes, with $h = 10$, $p_* = 0.1$, $p_{//} = 0.05$, $p_\lambda = 0.1$, and $\theta = 1$. We used the widely-used NITF (News Industry Text Format) DTD [9] as the input DTD of our XPath generator. All the algorithms were implemented in C++ and compiled using GNU C++ version 2.96. Experiments were conducted on 1.5 GHz Intel Pentium IV machines with 512 MB of main memory running Linux 2.4.18.

XSEARCH Efficiency. We evaluated the efficiency of the XSEARCH algorithm for search sets of different sizes. For this experiment, we considered search sets with unique subscriptions, that is, a given subscription does not appear more than once in a set. Indeed, in a given router, XSEARCH is used to determine the covering relationships between a given subscription and the subscriptions in the routing table, which are all unique.

For each search set, we generated $1,000$ additional subscriptions and, for each of them, we measured the time necessary to determine the subset of the subscriptions that cover, and are covered by, that subscription. For comparison purposes, we have also measured the efficiency of the XSEARCH algorithm against sequential execution of the containment algorithm of [5], which we call *Linear*.

Figure 4 shows the average search time of the XSEARCH algorithm. It appears clearly that $XSearch$ is extremely efficient. Even for very large search sets, we can expect an

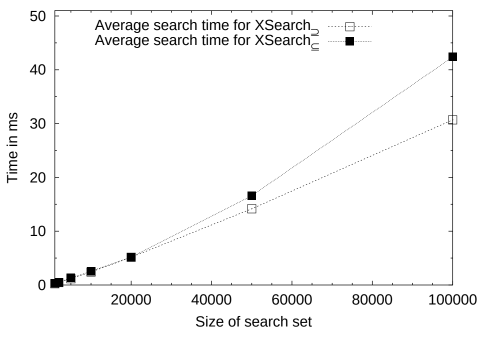


Figure 4: Average search time for the XSEARCH algorithm.

average search time of less than 50 ms. In comparison, the *Linear* algorithm yields to search times that are systematically more than two orders of magnitude higher. This result is not surprising, as the *Linear* algorithm needs to evaluate the entire subscription set R while $XSearch$ only searches through the factorization tree, which is much smaller by construction.

The second variant of the algorithm, $XSearch_\subseteq$, tends to be slightly less efficient than $XSearch_\supseteq$ for large consumer populations. We can explain this observation by the fact that, on average, the $XSearch_\subseteq$ algorithm necessitates deeper traversals of the factorization tree.

Size of search set	1,000	2,000	5,000	10,000
XSEARCH$_\supseteq$	0.23	0.45	1.17	2.41
XSEARCH$_\subseteq$	0.28	0.53	1.30	2.57

Table 1: Average search time of XSEARCH in ms.

One should note that, in practice, the sizes of the routing tables rarely exceed $1,000$ entries, even for very large consumer populations, thanks to subscriptions aggregation. For completeness, we show in Table 1 the absolute average search time of XSEARCH for search sets of small sizes, which are most relevant in the context of content-based routing.

$\|R\|$	1,000	2,000	5,000	10,000	20,000	50,000	100,000
$\sum_{s_i \in R} \|s_i\|$	7.6	15.8	42.1	88.1	183.3	481.8	998.6
$\|T(R)\|$	1.9	3.6	8.2	15.1	28.1	62.1	112.6

Table 2: Space requirements for a given subscription population R and its factorization tree $T(R)$, in thousands of nodes.

Space Efficiency. We have experimentally quantified the space requirements of the factorization tree with subscription sets of various sizes. The results in Table 2 confirm that the number of nodes in the factorization tree is indeed notably smaller than the sum of the nodes of the individual subscriptions.

6 Conclusion

We have described the subscription management techniques that we implemented in our XNET content routing network. These techniques rely on XSEARCH, an algorithm that determines the covering relationships between subscriptions, to efficiently process consumer registrations and cancellations. By capitalizing the performance of this algorithm, our content-based publish/subscribe system can maintain compact routing tables (for improved routing performance) while ensuring perfect routing (for bandwidth efficiency) at all time. Although described in the context of content-based routing and XPath, the XSEARCH algorithm can be readily used with similar subscription languages or to address different data management problems.

References

[1] R. A. Baeza-Yates and G. H. Gonnet. Fast text searching for regular expressions or automaton searching on tries. *Journal of the ACM*, 43(6):915–936, 1996.

[2] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, and D. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *Proceedings of ICDCS*, May 1999.

[3] A. Carzaniga, D. Rosenblum, and A. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.

[4] A. Carzaniga, M. J. Rutherford, and A. L. Wolf. A routing scheme for content-based networking. In *Proceedings of IEEE INFOCOM*, Mar. 2004.

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

[6] C.-Y. Chan, M. Garofalakis, and R. Rastogi. Re-tree: an efficient index structure for regular expressions. *The VLDB Journal*, 12(2):102–119, 2003.

[7] R. Chand and P. Felber. A scalable protocol for content-based routing in overlay networks. In *Proceedings of NCA*, Cambridge, MA, Apr. 2003.

[8] R. Chand and P. Felber. XNet: An XML Content Routing Network. Technical report, Institut EURECOM, 2004.

[9] I. P. T. Council. News Industry Text Format.

[10] G. Cugola, E. D. Nitto, and A. Fugetta. The JEDI event-based infrastructure and its application to the development of the opss wfms. *IEEE Transactions on Software Engineering*, 27(9):827–850, Sept. 2001.

[11] G. Miklau and D. Suciu. Containment and equivalence for an xpath fragment. In *Proceedings of PODS*, Madison, WI, June 2002.

[12] A. Tozawa and M. Hagiya. XML schema containment checking based on semi-implicit techniques. In *Proceedings of the Conference on Implementation and Application of Automata (CIAA)*, July 2003.

[13] W3C. XML Path Language (XPath) 1.0, Nov. 1999.

[14] S. Wu and U. Manber. Fast text searching: allowing errors. *Communications of the ACM*, 35(10):83–91, 1992.