# FoXtrot: Distributed Structural and Value XML Filtering

IRIS MILIARAKI and MANOLIS KOUBARAKIS, National and Kapodistrian University of Athens



Publish/subscribe systems have emerged in recent years as a promising paradigm for offering various popular notification services. In this context, many XML filtering systems have been proposed to efficiently identify XML data that matches user interests expressed as queries in an XML query language like XPath. However, in order to offer XML filtering functionality on an Internet-scale, we need to deploy such a service in a distributed environment, avoiding bottlenecks that can deteriorate performance. In this work, we design and implement FoXtrot, a system for filtering XML data that combines the strengths of automata for efficient filtering and distributed hash tables for building a fully distributed system. Apart from structural-matching, performed using automata, we also discuss different methods for evaluating value-based predicates. We perform an extensive experimental evaluation of our system, FoXtrot, on a local cluster and on the PlanetLab network and demonstrate that it can index millions of user queries, achieving a high indexing and filtering throughput. At the same time, FoXtrot exhibits very good load-balancing properties and improves its performance as we increase the size of the network.

Categories and Subject Descriptors: H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Information filtering*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems; Current awareness systems (selective dissemination of information–SDI); Performance evaluation*

General Terms: Algorithms, Design, Experimentation

Additional Key Words and Phrases: XML filtering, automata, load-balancing, distributed hash tables

**ACM Reference Format:**
Miliaraki, I. and Koubarakis, M. 2012. FoXtrot: Distributed structural and value XML filtering. ACM Trans. Web 6, 3, Article 12 (September 2012), 34 pages.
DOI = 10.1145/2344416.2344419 http://doi.acm.org/10.1145/2344416.2344419

## 1. INTRODUCTION

As the Web is growing continuously, a great amount of data is available to users, making it more difficult for them to discover interesting information by searching. For this reason, publish/subscribe systems, also referred to as information filtering systems, have emerged in recent years as a promising paradigm. In a publish/subscribe system, users express their interests by submitting a *continuous query* or *subscription*, and wait to be notified whenever an event of interest occurs or some interesting piece of information becomes available. Applications of such systems include popular notification services such as news monitoring, blog monitoring, and alerting services for digital libraries. Since XML is widely used for data exchange on the Web, a lot of research has focused on designing efficient and scalable XML filtering systems.

---

This is a revised and extended version of the paper by Miliaraki et al. [2008].
This work was supported by Microsoft Research through its Ph.D. Scholarship Programme.
Authors' address: I. Miliaraki and M. Koubarakis, Department of Informatics and Telecommunications, National and Kapodistrian University of Athens, Athens, Greece.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2012 ACM 1559-1131/2012/09-ART12 $15.00
DOI 10.1145/2344416.2344419 http://doi.acm.org/10.1145/2344416.2344419

In XML filtering systems, subscribers submit continuous queries, expressed in XPath/XQuery, asking to be notified whenever their queries are satisfied by incoming XML documents. In recent years, many centralized approaches like YFilter [Diao et al. 2003] and XTrie [Chan et al. 2002] have been presented for providing efficient filtering of XML data against large sets of continuous queries. However, in order to offer XML filtering functionality on an Internet-scale and avoid the typical problems of centralized solutions, such as single point of failure, lack of scalability, and network bottlenecks, this functionality should be offered in a distributed environment. Consequently, systems like XNet [Chand and Felber 2008] and ONYX [Diao et al. 2004] that implement distributed XML filtering have been proposed. The majority of the distributed approaches assume an overlay network with content-based routers responsible for forwarding XML data towards interested subscribers. For example, in the ONYX system, each broker keeps a broadcast tree for reaching all other brokers in the network, and uses a routing table for forwarding messages only to interested brokers. To achieve this, brokers in ONYX use instances of the YFilter engine [Diao et al. 2003] as routing tables.

Two important decisions in the above proposals is how to distribute queries among the brokers and what paths XML data follows in the network during filtering. Depending on these decisions, the brokers can suffer different amounts of load. The load of a broker includes indexing queries, filtering of incoming XML data, and delivering notifications to interested users whenever a query is matched. Consider for example the case of tree-based overlays where load imbalances occur, since the brokers closer to the root suffer more load either for routing queries or for forwarding XML data towards their destination. Unbalanced load can cause a performance deterioration to these tasks, especially as the size of the query set increases, incoming data arrives at a high rate or a large number of notifications is generated. As a result, part of the network becomes overloaded. In ONYX, the authors use a centralized component for assigning queries and data sources to the brokers of the network using criteria like topological distances and bandwidth availability in order to minimize latencies, but without actually dealing with load distribution. Other systems, like the one of Gong et al. [2005] do not deal at all with the amount of load suffered by each broker.

Load balancing in a distributed setting can be crucial for achieving high performance and scalability. With this in mind, we propose an alternative architecture that exploits the power of distributed hash tables (DHTs), a well-known class of structured overlay networks, to overcome the weaknesses of other proposals and develop a fully distributed load-balanced system. Our design allows us to apply simple yet effective load-balancing techniques using replication for achieving an equal distribution of load among the network peers.

Since automata and tree-based structures have proven to be highly efficient ways by many state-of-the-art XML filtering systems like YFilter [Diao et al. 2003] and XPush [Gupta and Suciu 2003] for indexing path queries, the main idea of our approach is to adopt such a technique and study how to implement it by exploiting the distributed setting of a DHT. For this purpose, we design and implement FoXtrot (***F**iltering **o**f **X**ML data on top of s**tr**uctured **o**verlay ne**t**works*), a system used for XML filtering on top of a network of peers organized using a DHT overlay. We propose to use a *nondeterministic finite automaton* (NFA) as in the state-of-the-art filtering engine, YFilter. We describe how to construct, maintain, and execute an NFA which encodes a set of XPath queries on top of a DHT. This *distributed* NFA is maintained by having peers responsible for overlapping fragments of the corresponding NFA. The size of these fragments is a tunable system parameter that allows us to control the amount of generated network traffic and load imposed on each peer.

Apart from using the distributed NFA for representing a set of queries and efficiently identifying XML documents that structurally match XPath queries, we also deal with the evaluation of value-based predicates (called *value matching*). Value matching is important because typical queries, apart from defining a structural path (e.g., */bib/article/citation*), also contain value-based predicates (e.g., */bib/article*[*@year* > 2007] */author*[*text*() = "*John Smith*"]). Depending on the selectivity of these predicates, the number of queries which are only structurally matched (i.e., false positives), might be large. For this reason, the benefit of using a filtering engine, for structural matching only, can be diminished. To the best of our knowledge, the only approach that deals explicitly with the evaluation of value-based predicates in a distributed environment is the XNet system [Chand and Felber 2008]. Value-based predicates are handled in XNet by associating each node of the tree structure used for organizing the queries with a set of predicates.

The main contributions of this article are the following.

—We design and implement a fully-distributed system, called FoXtrot, for efficient filtering of XML data on very large sets of XPath queries. To achieve this, we utilize the successful automata-based XML filtering engine, YFilter, distribute the automaton among the network peers, and design methods that exploit the inherent parallelism of an NFA. This way different peers participate in the filtering process by executing in parallel several paths of the NFA.

—We show that our approach overcomes the weaknesses of typical content-based XML dissemination systems built on top of meshes or tree-based overlays, while paying special attention to load balancing. The design of FoXtrot allows us to employ simple yet effective replication methods for achieving a balanced load distribution among the network peers. In addition, there is no need for any kind of centralized component to assign the queries to the network peers, since queries are distributed using the underlying DHT infrastructure.

—We demonstrate that, apart from structural matching, our system FoXtrot can also deal in an efficient way with value-based predicates. We briefly describe our different methods for value-matching and discuss our recent results on the topic [Miliaraki and Koubarakis 2010; Miliaraki 2011]. We select one of the proposed evaluation methods for inclusion in FoXtrot.

—We perform an extensive experimental evaluation in both the controlled environment, provided by a local cluster and PlanetLab which represents the real-world conditions of the Internet. We demonstrate that FoXtrot can index millions of user queries, achieving a high throughput of around 1000 queries per second in the local cluster, and outperforming other related systems. With respect to filtering, FoXtrot generates and disseminates more than 1500 notifications per second for the filtering scenarios we consider. We also show that our system exhibits scalability with respect to the network size, improving its performance as we add more peers to the network.

—We provide an extensive survey of related work ranging from centralized and distributed XML filtering methods to works that distribute several kinds of tree structures on top of DHTs.

Preliminary results of this research have appeared in our previous work [Miliaraki et al. 2008]. The current article revises that work and presents the following modifications, extensions, and additional contributions. First, while we initially designed and tested our methods using a simulated Chord network [Stoica et al. 2001], in this work we have fully implemented an XML filtering system called FoXtrot on top of Pastry DHT [Rowstron and Druschel 2001] using FreePastry release [2009]. Additionally, we provide a more detailed description of our algorithms including pseudocode. Even though we implement our system using an open-source implementation of Pastry,

our techniques are DHT-agnostic and any other implementation could be used instead. We also include a brief description of our different methods for combining structural and value XML filtering in a distributed way and discuss our recent results on the topic [Miliaraki and Koubarakis 2010]. Second, we study load-balancing techniques aiming to distribute in a uniform way the different loads of the network peers by extending and modifying the techniques presented in our previous work [Miliaraki et al. 2008]. Apart from the techniques that aim to balance storage load in specific DHT overlays, we design replication techniques based on the properties of the NFA structure which is distributed among the peers in FoXtrot. Third, with respect to the evaluation of FoXtrot, while we previously tested our algorithms using simulations, in this work we provide an extensive experimental evaluation of our implementation using the controlled environment of a local shared cluster and the worldwide testbed provided by the PlanetLab network. We include results demonstrating how various settings can impact the performance of FoXtrot and explain why this happens. In our previous study, we demonstrated the performance of our methods while varying the size of the indexed query set and the size of the network. In this work we also extend our evaluation by studying how the system parameter $l$, which tunes the size of the NFA fragments that each peer is aware of, affects the performance of the system and demonstrate the scalability of FoXtrot with respect to network size in a real environment. In addition, we study how the various characteristics of both the indexed queries and the documents being filtered (e.g., query depth, predicates per query, document depth) affect the performance of our system. Finally, we provide an extensive survey of related work, including centralized and distributed XML filtering approaches, related peer-to-peer systems, since FoXtrot is built on top of a DHT, and other approaches that distribute tree-like structures in such settings.

The organization of the article. In Section 2, we briefly describe the XML data model and the subset of XPath query language that we allow in FoXtrot, and also provide some background knowledge about nondeterministic finite automata and distributed hash tables. Section 3 describes in detail our methods for performing structural matching in FoXtrot; while Section 4 discusses the methods used for value matching. In Sections 5 and 6, we describe our experimental setting and present the results of our evaluation. Section 7 provides a survey of related work in the area of XML filtering and distributed publish/subscribe systems in general. Finally, Section 8 concludes the article and discusses future work.

## 2. BACKGROUND

In this section we give a short introduction to the XML data model, the subset of XPath we allow, nondeterministic finite automata, and distributed hash tables.

### 2.1. XML and XPath

An XML document can be represented using a rooted, ordered, labeled tree where each node represents an element or a value and each edge represents relationships between nodes such as an element–subelement relationship. Element nodes may contain attributes which describe their additional properties or textual data.

XPath [Clark and DeRose 1999] is a language for navigating through the tree structure of an XML document. XPath treats an XML document as a tree and offers a way to select paths of this tree. Each XPath expression consists of a sequence of *location steps*. We consider location steps of the following form:

$$axis\ nodetest\ [predicate_1] \ldots [predicate_n]$$

where *axis* is a child (/) or a descendant (//) axis, *nodetest* is the name of the node or the wildcard character "*", and $predicate_i$ is a predicate in a list of one or more predicates

used to refine the selection of the node. Each predicate is either an *attribute predicate* of the form [*attr op value*] where *attr* is an attribute name, *value* is an attribute value and *op* is one of the basic logical comparison operators $\{=, >, >=, <, <=, <>\}$ or a *textual predicate* of the form [*text*() *op value*] where *value* is a string value and *op* is a string operator, as defined in Clark and DeRose [1999].

A *linear path query* $q$ is an expression of the form $l_1 l_2 \ldots l_n$, where each $l_i$ is a location step. In this article, queries are written using this subset of XPath, and we will refer to such queries as *path queries* or *XPath queries* interchangeably. Example path queries for a bibliographic database are

$Q_1$: `/bib/phdthesis[@published=2005]/author[@nationality=greek]`
which selects PhD theses published in year 2005 by Greek authors.
$Q_2$: `/bib/*/author[text()="John Smith"]`
which selects any publication of author John Smith.

### 2.2. Nondeterministic Finite Automata

A nondeterministic finite automaton (NFA) is a 5-tuple $A = (Q, \Sigma, \delta, q_0, F)$, where $Q$ is a finite set of states, $\Sigma$ is a finite set of input symbols, $q_0 \in Q$ is the *start state*, $F \subseteq Q$ is the set of *accepting states*, and $\delta$, the *transition function*, is a function that takes as arguments a state in $Q$ and a member of $\Sigma \cup \{\epsilon\}$ and returns a subset of $Q$ [Hopcroft et al. 2000]. The *language* $L(A)$ of an NFA $A = (Q, \Sigma, \delta, q_0, F)$ is $L(A) = \{w \mid \hat{\delta}(w, q_0) \cap F \neq 0\}$. $L(A)$ is the set of strings $w$ in $\Sigma \cup \{\epsilon\}$ such that $\hat{\delta}(q_0, w)$ contains at least one accepting state, where $\hat{\delta}$ is the extended transition function constructed from $\delta$. Function $\hat{\delta}$ takes a state $q$ and a string of input symbols $w$, and returns the set of states that the NFA is in, if it starts in state $q$ and processes the string $w$. Typically, NFAs allowing $\epsilon$-transitions (transitions without receiving an input symbol) are called $\epsilon$-NFAs. However, throughout this article we will use the term NFA for actually referring to an $\epsilon$-NFA.

Any path query can be transformed into a regular expression, and consequently there exists an NFA that accepts the language described by this query [Hopcroft et al. 2000]. Following YFilter [Diao et al. 2003], for a given set of path queries, we will construct an NFA $A = (Q, \Sigma, \delta, q_0, F)$, where $\Sigma$ contains element names and the *wildcard* (*) character, and each path query is associated with an accepting state $q \in F$. An example of this construction is depicted in Figure 1. The figure also shows the different location steps and the corresponding NFA fragments.

### 2.3. Distributed Hash Tables

DHTs have emerged as a promising way of providing a highly efficient, scalable, robust, and fault-tolerant infrastructure for the development of distributed applications. Although there have been many proposals of DHTs such as Chord [Stoica et al. 2001], Pastry [Rowstron and Druschel 2001], and CAN [Ratnasamy et al. 2001] which differ in their technical details, but all try to solve the following *lookup* problem: given a data item $x$ stored in a network of peers, find $x$. The core idea in all different DHTs is to solve this search problem by offering a kind of distributed hash table functionality where data items are uniquely identified by keys and DHT peers cooperate to store these keys. This is achieved by providing a lookup($k$) operation returning a pointer to the DHT node responsible for key $k$. In Pastry [Rowstron and Druschel 2001], each peer and each data item is assigned a unique $m$-bit identifier. The identifier of a peer can be computed by hashing its IP address and is used to indicate its position in a circular identifier space ranging from 0 to $2^m - 1$. For data items, we first have to compute a *key* and then hash this key to obtain the identifier. Pastry routes messages to the peer whose identifier is numerically closest to the given key, using a technique called prefix

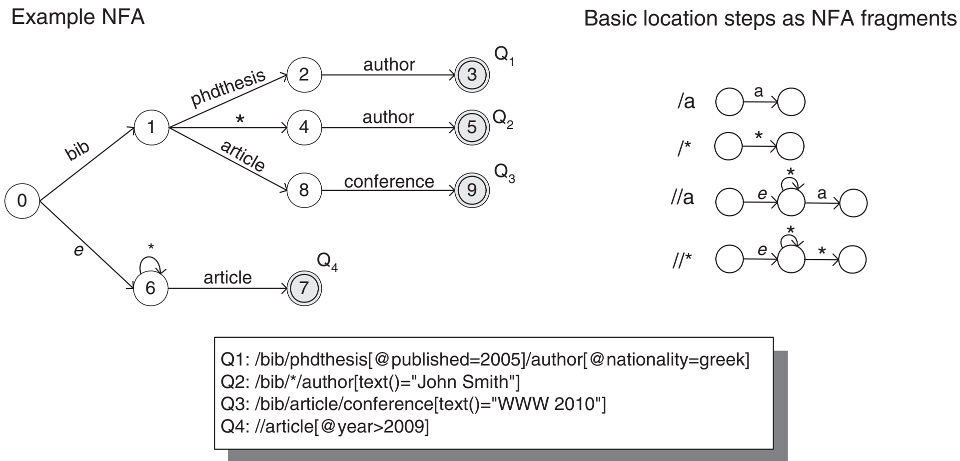



Fig. 1. An example NFA constructed from a set of XPath queries.

routing. Such requests can be done in $O(\log n)$ steps, where $n$ is the number of nodes in the network.

In the rest of the article we use Pastry as the underlying DHT and implement our system using FreePastry release [2009]. However, our techniques are DHT-agnostic; they can be implemented on any DHT that offers the standard lookup operation. For more details on DHTs and peer-to-peer systems in general, the interested reader can see the survey of Lua et al. [2005] or other more detailed studies on DHTs like the one from Balakrishnan et al. [2003].

## 3. STRUCTURAL MATCHING

In this work we design and implement FoXtrot, a system for filtering XML data against a set of XPath queries. FoXtrot supports queries that consist of both structural and value predicates, as described in our data model. In this section we focus on the methods used for structural matching. Automata and tree-based structures have proven to be efficient ways for indexing path queries by many state-of-the-art XML filtering systems like YFilter [Diao et al. 2003], XTrie [Chan et al. 2002], XPush [Gupta and Suciu 2003], and Index-Filter [Bruno et al. 2003]. For this reason, we decided to use an NFA-based model, similar to the one used in YFilter, for indexing path queries in our system and performing structural matching. The NFA is constructed from a set of XPath queries and used as a distributed matching engine that scans incoming XML documents and discovers matching queries. In this section we describe in detail how the NFA corresponding to a set of XPath queries is constructed, maintained, and executed by the network peers for providing XML filtering functionality in FoXtrot. Value-matching is briefly discussed in the next section, while our methods for load-balancing are described in Section 6 before our experimental evaluation.

### 3.1. Distributing the NFA

The NFA corresponding to a set of path queries is essentially a tree-like structure that needs to be traversed both for indexing a query during NFA construction and for finding matches against incoming XML data during NFA execution. In FoXtrot, we distribute an NFA on top of Pastry and provide efficient ways of supporting these two basic operations. Our main motivation for distributing the automaton derives from the nondeterministic nature of NFAs that allows them to be in several states at the same

time, resulting in many different parallel executions. We also preferred to use an NFA instead of its equivalent DFA for reducing the number of states. Since we distribute the NFA on top of a Pastry network, we use the term *distributed NFA* to refer to it.

The distribution of the NFA among the network peers in FoXtrot is done at the level of the NFA states by assigning them to the network peers, as follows. Each state $q_i$ along with every other state included in $\hat{\delta}(q_i, w)$, where $w$ is a string of length $l$ included in $\Sigma \cup \{\epsilon\}$, is assigned to a single peer in the network. Note that $l$ is a parameter that determines how large a part of the NFA is the responsibility of each peer. If $l = 0$, each state is indexed only once at a single peer, with the exception of states that are reached by an $\epsilon$-transition, which are also stored at the peers responsible for the state that contains the $\epsilon$-transition. Recall that the $\epsilon$-transition represents a transition that can be followed without receiving an input symbol. For larger values of $l$, each state is stored at a single peer along with other states reachable from it by following a path of length $l$. This results in storing each state at more than one peer. Therefore, peers store overlapping fragments of the NFA, and parameter $l$ characterizes the size of these fragments. Apart from parameter $l$, we also employ replication techniques to achieve a balanced load in the system. These techniques are complementary to the methods we describe here, and are discussed in detail in Section 6.

To determine which peer will be responsible for each state, we uniquely identify each state with a key. The *responsible peer* for state with key $k$ is the peer whose identifier is numerically closest to *Hash*(*k*), where *Hash*() is the DHT hash function. The key of an automaton state is formed by the concatenation of the labels of the transitions included in the path leading to the state. For example, the key of state 2 in Figure 1 is the string '*start*'+'*bib*'+'*phdthesis*', the key of the start state is '*start*' and state 9 has key '*start*'+'$', since we choose to represent $\epsilon$-transitions using character $. Operator + is used to denote the concatenation of strings. Each peer $p$ keeps a hash table, denoted by *p.states*, which contains the states assigned to $p$ indexed by their keys. For each state *st* included in *states* we keep the transitions from *st*, including potential self-loops, and, if *st* is an accepting state, we also keep the identifiers and the subscribers of the relevant queries. Recall that a query matches a document if, during the execution of the NFA, the accepting state for that query is reached. We denote the list containing these queries as *st.queries*.

*Example.* Figure 2 illustrates how an example NFA is distributed on top of Pastry when $l = 1$. We assume a network of 9 peers and depict where each state is stored on the Pastry ring. Notice that state 10 is included in $P7.states = [0, 1, 9, 10]$ because the $\epsilon$-transition does not contribute to the specified length $l$. To ease readability, we use unique integers to represent the state keys.

### 3.2. Constructing the Distributed NFA

In this section we describe how we achieve the distribution of the NFA among the DHT peers in FoXtrot as queries arrive in the system and the automaton is incrementally constructed.

To help the reader understand this process, we first describe how the NFA is constructed without considering the fact that the states are distributed and will be stored at different peers. This process is identical to the construction process in the centralized environment of YFilter. A location step in a query can be represented by an NFA fragment [Diao et al. 2003]. The NFA for a path query can be constructed by concatenating the NFA fragments of the location steps it consists of, and making the final state of the NFA the accepting state of the path query. Inserting a new query into an existing NFA requires us to combine the NFA of the query with the already existing one. So to insert a new query represented by an NFA $S$ to an existing NFA $R$, we start

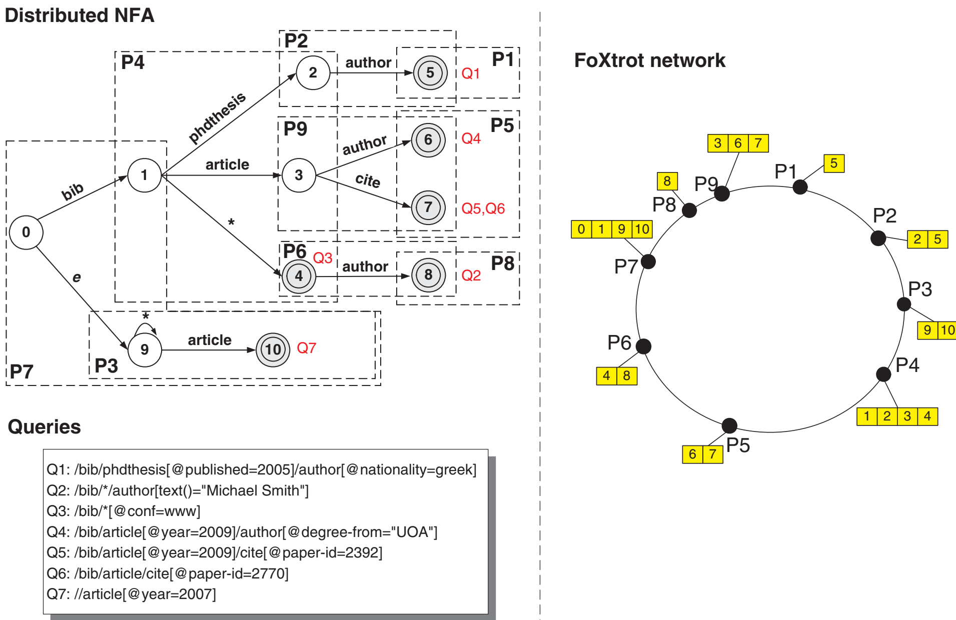



Fig. 2. Distributing an NFA in FoXtrot ($l = 1$).

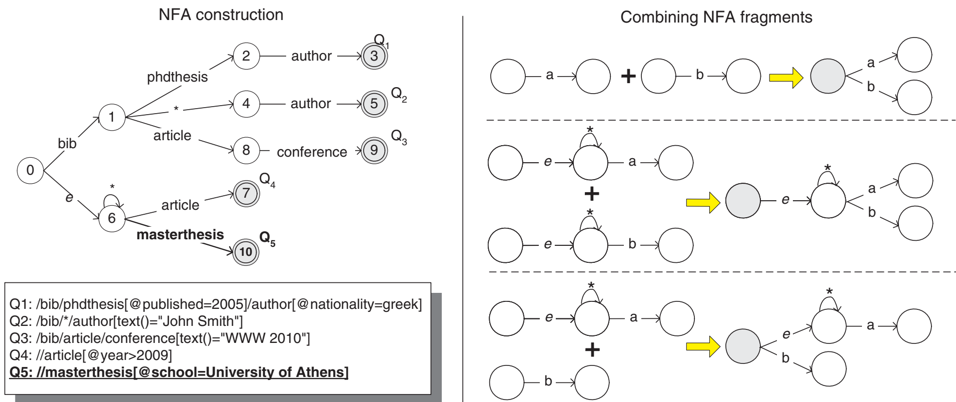



Fig. 3. NFA construction.

from the common start state shared by $R$ and $S$, and we traverse $R$ until we either reach the accepting state of $S$ or a state for which there is no transition that matches the corresponding transition of $S$. If the latter happens, a new transition is added to that state in $R$. Formally, if $L(R)$ is the language of the NFA already constructed by previously inserted queries, and $L(S)$ is the language of the NFA of the query being indexed, then the resulting NFA has language $L(R) \cup L(S)$. Examples of how different NFA fragments are combined are shown in Figure 3. We also depict the NFA of Figure 1 after inserting query $Q5$.

Let us now describe how we traverse the distributed NFA for inserting a query $q$. The main idea is that whenever we want to visit a particular NFA state during indexing $q$, we first discover and contact the peer responsible for that state. If the state does not

**ALGORITHM 1:** `IndexQuery()`: *Indexing a query*

```
1  procedure peer.IndexQuery(q, d, st, sid, l, firstCall)
2      if peer.states does not contain st then
3          add st to peer.states;
4      else
5          st := peer.states.get(st.key);
6      if d == q.length then
7          add q to st.queries;
8      else
9          t := transition label of q at depth d;
10         if no transition exists labeled t from st to st′ then
11             add transition labeled t from st to st′;
12             if t == $ then
13                 st′ := st.getTransition(t);
14                 st′.selfChild := true;
15
16         else  st′ := st.getTransition(t);
17         if t == $ then
18             t′ := transition label of q at depth d + 1;
19             if no transition exists labeled t′ from st′ to st″ then
20                 add transition labeled t′ from st′ to st″;
21
22         if firstCall is true then
23             nextPeer := Lookup(st′.key);
24             nextPeer.Route(IndexQueryMesg(q, d+1, st′, sid));
25         else
26             if l > 0 then
27                 if t == $ then
28                     peer.IndexQuery (q, d+1, st′, sid, l, false);
29                 else
30                     peer.IndexQuery (q, d+1, st′, sid, l − 1, false);
31
32
33
34
```

exist, the relevant peer creates it. If the state exists, then the peer may need to update it by adding a new transition. The exact steps followed are depicted in Algorithm 1. Algorithms in this article are described using a notation where *p.Proc*() means that peer *p* receives a message and initiates execution for procedure *Proc*().

Suppose *s* is the subscriber peer for query *q*. Using Pastry, peer *s* sends a message `IndexQueryMesg`($q, d, st, sid$) to peer *r*, where *q* is the query being indexed in the form of an NFA, *d* is the current depth of the query NFA reached, *st* is the state at this depth, and *sid* is the identifier of the subscriber peer. Initially $d = 0$, *st* is the start state, and *r* is the peer responsible for it. Starting from this peer, each peer *r* that receives an `IndexQueryMesg` message, executes locally the corresponding procedure `IndexQuery`($q$, $d$, $st$, $sid$, $l$, $firstCall$), where $l$ is value of the system parameter $l$ and $firstCall$ is a Boolean parameter initially true. If $l$ is larger than 0, then *r* recursively calls this procedure and stores locally the additional states as defined by $l$. To distinguish between the first call of the procedure and the recursive calls, we use parameter $firstCall$.

The details of the local procedure `IndexQuery` executed at each peer for a state *st* are as follows. At first, the peer checks whether *st* is already stored locally. Recall that each NFA state is identified by a sequence of transition labels. If *st* is not stored locally, it creates it (lines 2-5). If *st* is the accepting state of *q*, *q* is inserted in the list *st.queries* and execution ends (lines 6,7). At this point, the responsible peer can notify the subscriber peer that *q* is successfully indexed. Otherwise, query indexing continues with the next state. Let *t* be the label of the transition from state *st* to a target state *st*′. Then, if there is no such transition from *st*, a new transition is added

from state $st$ to $st'$ with label $t$ (lines 10-11). If this transition is an empty transition, then a self-loop transition is also added to state $st'$ to represent a "//" step (lines 12-15). In addition, we need to fix the local transition table of the next state $st'$ (lines 17-20). Finally, indexing proceeds by sending a new `IndexQueryMesg`($q$, $d+1$, $st'$, $sid$) message to the next responsible peer (i.e., the peer responsible for $st'$) increasing query depth by 1 (lines 22-24). If $l$ is larger than 0, the procedure is called recursively $l$ additional times by the peer to store locally the extra states (lines 25-30).

Constructing the NFA as described above, requires sending as many `IndexQueryMesg` messages as the number of states in the NFA of query $q$. The number of messages that travel through the network during the construction of the NFA is independent of the value of $l$, while $l$ affects the time spent by each peer during the local processing of the indexing request.

### 3.3. Executing the Distributed NFA

In this section we describe how we execute the distributed NFA during XML filtering for discovering matching queries. We first describe how the NFA is executed without considering the fact that the states are distributed. This process is similar to the execution process in the centralized environment of YFilter. Then, we describe two different methods for executing the distributed NFA, namely the iterative and the recursive methods.

The NFA execution proceeds in an event-driven fashion. As the XML document is parsed, the events produced are fed, one event at a time, to the NFA to drive its transitions. Parsing is performed using a SAX parser that produces events of the following types: *StartOfElement, EndOfElement, StartOfDocument, EndOfDocument*, and *Text*. The nesting of elements in an XML document requires that when an *EndOfElement* event is raised, the NFA execution should backtrack to the states it was in when the corresponding *StartOfElement* was raised. For achieving this, YFilter maintains a stack, called the *runtime stack*, during the execution of the NFA. Since many states can be active at the same time in an NFA, the stack is used for tracking multiple active paths. The states placed on the top of the stack will represent the *active states*, while the states found during each step of execution after following the transitions caused by the input event, will be called the *target states*. Execution is initiated when a *StartOfDocument* event occurs and the start state of the NFA is pushed into the stack as the only active state. Then, each time a *StartOfElement* event occurs for an element $e$, all active states are checked for transitions labeled with $e$, *wildcard*, and $\epsilon$-transitions. In case of an $\epsilon$-transition, the target state is recursively checked one more time. All active states containing a *self-loop* are also added to the target states. The target states are pushed into the runtime stack and become the active states for the next execution step. If an *EndOfElement* event occurs, the top of the runtime stack is popped and backtracking takes place. Execution proceeds in this way until the document has been completely parsed or the stack becomes empty.

As with the YFilter, for executing the distributed NFA in FoXtrot we need to maintain a stack containing the states for backtracking. For each active state we need to retrieve all target states reached after feeding the corresponding parsing event to the NFA. Since states are distributed among the network peers, at each step of the execution, the relevant parsing event should be forwarded to the peers responsible for the active states. We identify two ways for executing the NFA: the first proceeds in an iterative way, while the other executes the NFA in a recursive fashion.

*3.3.1. Iterative Method.* In this method, the publisher peer is responsible for parsing the document, maintaining the runtime stack, and forwarding the parsing events to the responsible peers to get the target states and continue execution. As a result,

**ALGORITHM 2:** `PublishIterative()`: *Publishing an XML document - Iterative way*

```
1  procedure peer.PublishIterative(doc)
2      parsingEvents = parse(doc);
3      publisherId = peer.getId();
4      if l == 0 then
5          pathLength := 1;
6      else
7          pathLength := l;
8      firstPeer := Lookup("start");
9      Mesg := GetStateMesg ("start") ;
10     startState := firstPeer.Route(Mesg);
11     add startState to activeStates;
12     while parsingEvents.size ! = 0 do
13         initialize event,currentEvents;
14         while currentEvents.size < pathLength do
15             event = parsingEvents.getNext() ;
16             if event is endElement then
17                 break;
18             else
19                 add event to currentEvents;
20
21         foreach state in activeStates do
22             responsiblePeer := Lookup(state.key);
23             Mesg := GetTargetStatesMesg (state, currentEvents, publisherId));
24             targetStates.add(responsiblePeer.Route(Mesg);
25         foreach state in targetStates do
26             if state.queries > 0 then
27                 notify interested subscribers;
28
29         runtimeStack.push(targetStates);
30         if parsingEvents.getNext() is endElement then
31             runtimeStack.pop();
32         activeStates := runtimeStack.getTopElement();
```

the execution of the NFA proceeds in a similar way as described previously, with the exception that the target states cannot be retrieved locally but need to be retrieved from other peers of the network. Also since parameter $l$ allows peers to keep larger fragments of the NFA, a peer responsible for an active state during execution can exploit the whole relevant NFA fragment kept locally (i.e., the NFA fragment beginning at that state) and perform several subsequent expansions. Algorithm 2 describes the actions required by the publisher peer.

The publisher peer $p$ publishes a document by following the steps described in procedure `PublishIterative`(*doc*) where *doc* is the XML document being published. At first, $p$ parses the XML document and stores the corresponding parsing events in a list (line 2). Then, it initializes a variable called *pathLength* using the value of parameter $l$. If $l$ is equal to 0, then, along with each active state, the peer will send a single document element (from the parsing events) to the responsible peer. Else, if $l$ is greater than 0, the publisher peer will send more than one element with each active state, since the responsible peer will be able to perform more expansions (lines 4-7). At first $p$ communicates with the peer responsible for the start state to retrieve it and adds it to the active states to initiate NFA execution (lines 8-11). Next, peer $p$ begins reading the parsing events and inserts them in a list called *currentEvents* until either *pathLength* elements are inserted or an *EndElement* event is read (lines 12-20). Then, $p$ sends a `GetTargetStatesMesg` message to each peer responsible for an active state. During the first iteration, only the start state is active. Each responsible peer proceeds with the expansion of the relevant states and returns the corresponding target states back to the publisher. Again, depending on the value of $l$, each peer may perform multiple

**ALGORITHM 3:** `PublishRecursive()`: *Publishing an XML document using recursive method*

```
1 procedure peer.PublishRecursive(doc)
2     enrichedEvents := constructIndex(doc.parsingEvents);
3     firstPeer := Lookup("start");
4     currentIndex := 0;
5     parentIndex := -1;
6     Mesg := RecExpandStateMesg ("start", enrichedEvents, 0, −1) ;
7     firstPeer.Route (Mesg);
```

expansions by itself. The states are stored in the list *targetStates* (lines 21-24). Next, for each target state, $p$ checks whether there are any queries matched and notifies interested subscribers (lines 25-28). The target states are pushed on the runtime stack and become the active states for the next iteration of the execution (line 29). If the next parsing event is an *EndElement* event, the stack is popped (lines 30-31). Execution continues until the document has been completely parsed or the runtime stack becomes empty.

The iterative method imposes the majority of the load on the publisher peer, which is responsible for contacting several network peers and retrieving the states that are not locally stored. We expect and demonstrate experimentally that the iterative method will perform poorly, since the publisher peer can become a bottleneck, we presented it here to assist the reader in understanding the details of the recursive method.

*3.3.2. Recursive Method.* There can exist multiple active paths during NFA execution where each active path consists of a chain of states, starting from the start state and linking it to the target states. We design this method by exploiting the fact that these active paths are independent and can be executed in parallel by different peers. We achieve this as follows. The publisher peer forwards the XML document to the peer responsible for the start state to initiate the execution of the NFA. The execution continues recursively, with each peer responsible for an active state continuing the execution. Notice that the run-time stack is not explicitly maintained in this case, but it implicitly exists in the recursive executions of these paths. The execution of the NFA is parallelized in two cases. The first case is when the input event processed has siblings with respect to the position of the element in the tree structure of the XML document. In this case, a different execution path will be created for each sibling event. The second case is when more than one target state results from expanding a state. Then, a different path is created for each target state, and a different peer continues the execution for each such path. To efficiently check structural relationships between elements, the publisher peer enriches the parsing events *StartOfElement* and *EndOfElement* with a positional representation. Specifically, the events are enriched with the position of the corresponding element with a pair (*L*:*R*,*D*), where $L$ and $R$ are generated by counting tags from the beginning of the document until the start tag and the end tag of this element, and $D$ is its nesting depth. The publisher peer is responsible for enriching the parsing events. This representation was introduced by Consens and Milo [1994] and it requires an additional pass over the XML document.

Peer $p$ publishes the XML document *doc*, following the steps described in Algorithm 3. First, $p$ enriches the parsing events using a positional representation to enable efficient checking of structural relationships (line 2). Then, peer $p$ sends a message, `RecExpandStateMesg`(*'start'*, *enrichedEvents*, *currentIndex*, *parentIndex*), to peer $r$ which is responsible for the start state, where *enrichedEvents* is the list with the enriched parsing events of the XML document; *currentIndex* refers to the event that needs to be processed next (in this case 0 refers to the first element) and *parentIndex*

**ALGORITHM 4:** `RecExpandState()`: *Recursively expand states at each execution path - Recursive way*

```
1  procedure peer.RecExpandState(stateKey, enrichedEvents, currentIndex, parentIndex)
2      st := peer.states.get(stateKey);
3      add st to activeStates;
4      if l == 0 then pathLength := 1 else pathLength := l;
5      elementsProcessed := 0;
6      while elementsProcessed < pathLength && enrichedEvents.size != 0 do
7          currEvent = enrichedEvents.getNext();
8          if currEvent.isEndElement() then break;
9          if currEvent.hasSiblings() then
10             siblings := siblings of currEvent;
11             foreach siblingEvent in siblings do
12                 compute targetStates from each st in activeStates for input siblingEvent;
13                 foreach state in targetStates do
14                     if state.queries > 0 then notify interested subscribers;
15             break;
16         else
17             siblings := currEvent;
18             compute targetStates from each st in activeStates for input currEvent;
19             foreach state in targetStates do
20                 if state.queries > 0 then notify interested subscribers;
21         activeStates := targetStates ;
22         elementsProcessed++;
23     for i=0 to siblings.size do
24         currEvent := siblings.get(i);
25         nextEvent := siblings.get(i+1);
26         if nextEvent is endElement then continue;
27         nextIndex := nextEvent.getIndex();
28         nextParentIndex := currEvent.getIndex();
29         foreach nextState in targetStates do
30             nextPeer = Lookup(nextState.key);
31             Mesg := RecExpandStateMesg(nextState.key, enrichedEvents, nextIndex, nextParentIndex)
               nextPeer.Route(Mesg);
```

refers to its parent event (lines 3-7). At first, *parentIndex* is $-1$, since the root element of the document has no parent element.

When a peer receives a `RecExpandStateMesg` message, it executes the local procedure `RecExpandState`, depicted in Algorithm 4. First, peer $r$ retrieves *st* from its local store and adds it to a list containing the active states of the execution (lines 2-3). Similar to the iterative approach, $r$ initializes a variable, called *pathLength*, using the value of $l$ (line 4). If $l$ is equal to 0, then $r$ can perform a single expansion by itself. Else, if $l$ is greater than 0, $r$ can perform multiple expansions. Peer $r$ also keeps the number of the elements it has already processed, initially 0 (line 5). Next, $r$ begins the execution of the NFA in the relevant path starting with state *st* until either it performs the relevant number of expansions or it reaches the end of the document being filtered. If the corresponding element has no siblings, $r$ computes the expansions by itself in this single execution path and notifies the interested subscribers (lines 16-20). If the element has siblings, then it computes separately the expansions for each different sibling (lines 6-15). Suppose $e_1, \ldots, e_s$ are the sibling events and $TS(e_1), \ldots, TS(e_s)$ represent the sets with the target states computed by each event. These target states may have been computed either after a single expansion or after multiple expansions. Peer $r$ will forward $\sum \mid TS(e_i) \mid$ different `RecExpandStateMesg` messages, one for each of the different execution paths (lines 23-31). The execution for each path continues until the document fragment has been completely parsed. Peers that participate in

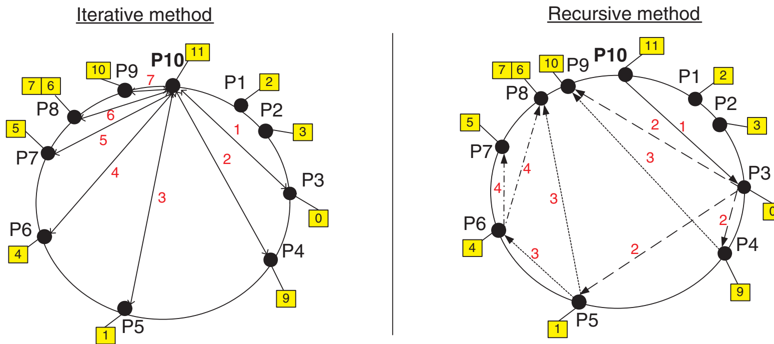



Fig. 4. Executing the distributed NFA.

the execution process are also responsible for notifying the subscribers of the satisfied queries.

Note that the recursive method assumes that the XML document being filtered is relatively small, and this is the reason for deciding to forward the whole document at each step of execution. In realistic scenarios, XML documents are usually small, as discussed by Barbosa et al. [2006]. However, in the case we want to filter larger XML documents, our method could be adjusted so that we forward smaller fragments of the document. This process is typically referred to as XML message transformation, and has been studied in related systems like ONYX [Diao et al. 2004].

*Example.* We demonstrate using an example of how peers communicate during the execution of the NFA in FoXtrot in both of the above cases. As Figure 4 illustrates, peer $P10$ is the publisher of an XML document. When the iterative method is used, all communications are initiated by the peer $P10$, which contacts 7 different peers and retrieves the corresponding states. With the recursive method, execution begins with $P10$ contacting $P3$, which is responsible for the start state. $P3$, continues execution by forwarding the corresponding filtering requests to peers $P5$ and $P9$. Then, filtering is continued in parallel by $P5$ and $P9$. In this example, we omit the details of the execution and focus on the sequence of the different communications occurring among the peers to demonstrate how these two methods compare.

## 4. VALUE MATCHING

In the previous section, we described how structural matching is performed in FoXtrot. While our approach and other similar approaches that employ automata or similar indices have been used with success for representing a set of queries and identifying XML documents that structurally match XPath queries, little attention has been paid to the evaluation of value-based predicates, especially in distributed settings. Consider for example query $q$:$/bib/article[@conf = WWW]/author[text() =$"*John Smith*"], which selects the articles of the author "John Smith" published in a WWW conference. Filtering incoming XML data against this query requires checking whether the data structurally matches the query and also whether the value-based predicates of the query are satisfied. This can be an important problem because, depending on the selectivity of these predicates, the number of queries which are only structurally matched (i.e., false positives), might be large. Our goal is to design a system that scales with respect to both the number of the queries indexed and the number of the predicates included in the queries. For this reason, we support techniques for dealing with the evaluation of value-based predicates together with structural matching in FoXtrot. In this section, we briefly consider

different ways for achieving this. The following techniques are described in detail in our recent work [Miliaraki and Koubarakis 2010; Miliaraki 2011].

Our first technique evaluates predicates after performing structural matching. Such a technique operates in a top-down fashion, processing incoming XML documents from the root towards their leaves. We use the distributed NFA to identify the subset of queries that structurally match incoming XML documents, and then evaluate the predicates of this subset of queries. Hence, this method evaluates predicates after the execution of the NFA. Since in FoXtrot structural matching is performed in parallel by multiple peers, each of these peers identifies a different subset of structurally-matched queries. Whenever a peer identifies such a set, it is also responsible for the predicate evaluation. We refer to this method as *top-down evaluation*.

To overcome possible shortcomings of top-down evaluation caused by spending too much effort on structurally matching queries with predicates that are not satisfied by incoming XML data, we also propose the following optimization. By using a compact summary of predicate information, we stop the execution of the distributed NFA (i.e., *prune* this execution path) whenever we can deduce that no match can be found if the execution continues. At each step of the execution, we consider that a part of the distributed NFA has been revealed while the rest part is not. We utilize Bloom filters for summarizing these NFA fragments with respect to the predicates they contain. Then, we decide whether or not we will continue execution by consulting these filters. Note that this method is only applicable to equality predicates, and we refer to it as *top-down evaluation with pruning*.

Following a widely used strategy from relational query optimization, where selections are applied as early as possible, we can check the value-based predicates before proceeding with the structural matching following a bottom-up approach. Such an approach evaluates XML documents in a bottom-up way, since in a tree representation, element values are placed in the leaves of the tree. In contrast to the other methods, where the indexing of the queries is done using the distributed NFA, a different indexing algorithm is required in this case. To first discover queries that contain specific predicates, indexing is based on these predicates. Such an indexing algorithm resembles work presented for information filtering (IF) on top of DHTs, including the work of Tryfonopoulos et al. [2005], where queries are expressed using a simple attribute-value data model and attribute values are used to map queries to peer identifiers. We refer to this method as *bottom-up evaluation*. A drawback of this approach is that, even though the heuristic of pushing selections early works well in the case of relational query processing, in our case peers may put a lot of effort in evaluating predicates for queries whose structure may not be matched later on.

Furthermore, considering that XPath queries consist of distinct steps and each step may be associated with one or more value-based predicates, we can perform, at each step of the NFA execution, structural matching simultaneously with predicate evaluation. Therefore, in this case we evaluate predicates during the NFA execution. We consider that the latter approach performs XML filtering in a step-by-step fashion.

Finally, since in our case the XPath queries are indexed using an NFA, we could perform predicate evaluation directly with the automaton by adding extra transitions for the predicates. An expected drawback of such a method comes from the fact that the elements in a set of XPath queries represent a rather small set, since they are constrained by the schema, while the values of the predicates may form a really large set. This could result in a huge increase of the NFA states and, at the same time, destroy the sharing of path expressions for which the NFA was selected to begin with. For this reason, we have not studied this method any further.

In our recent study [Miliaraki and Koubarakis 2010], we compared the above methods, namely top-down, top-down with pruning, bottom-up, and step-by-step, by fully

implementing them in our system, FoXtrot. The top-down method outperforms the others in terms of both network traffic and filtering latency. Since the emphasis of the present article is on structural matching, we consider the detailed comparison of the former methods outside of our scope, and point the reader to our recent study for more details. With respect to our experimental evaluation of FoXtrot, which follows, value matching is performed using one of the preceding methods, namely, top-down evaluation with pruning.

## 5. EXPERIMENTAL SETUP

In this section, we describe our experimental setting. We implemented FoXtrot in Java using FreePastry release [2009]. We ran our experiments in two different environments, the worldwide testbed for large-scale distributed systems provided by the PlanetLab network (http://www.planet-lab.org/) and a local shared cluster (http://www.grid.tuc.gr/).

### 5.1. Network setup

In the case of PlanetLab, we used 396 nodes that were available and lightly loaded at the time of the experiments. Note that PlanetLab nodes are geographically distributed among four continents and shared by many users. We also ran our experiments on a cluster that consists of 41 computing nodes connected by a Gigabit ethernet connection (1000Mbps). Each node is a server blade machine with two processors at 2.6GHz and 4GB memory. In this case, we used 28 of these machines, running up to 4 peers per machine, that is, 112 peers in total.

### 5.2. Document and Query Generation

We generated two different synthetic data sets using the IBM XML Generator [1999], namely, the mixed dataset and the NITF dataset. Each set consisted of 1000 documents. The mixed dataset is created using a set of 10 DTDs, including DBLP DTD, NITF (News Industry Text Format) DTD, ebXML DTD (Electronic Business using eXtensible Markup Language) and the Auction DTD from the XMark benchmark [XMark 2001]. Using this dataset, we study the performance of our approach in a realistic scenario where users subscribe to FoXtrot to receive notifications concerning various interests of theirs (e.g., information about scientific papers and news feeds). The second dataset is created using only the NITF DTD, which has been used in many works [Chan et al. 2002; Diao et al. 2003; Hou and Jacobsen 2006]. The NITF DTD allows 123 different element tags, 513 attributes, and represents an interesting case where a large fraction of XML elements are allowed to be recursive. The average document size for the mixed dataset is equal to 37.8 Kb, and for the NITF workload is 15 Kb.

The same DTDs were used to generate different sets of $10^6$ path queries with varying characteristics using the XPath generator available in the YFilter release [2004]. Each query set contained only distinct queries, in other words, there are no duplicates. In a realistic scenario where users share interests, such a query set can represent the interests of millions of users.

The values of the parameters used for generating both the document and the query sets are shown in Table I. The *depth of a path query* refers to the number of location steps contained in the query. The *depth of an XML document* is the longest nesting of an element appearing in the document. Note that queries also contain a number of value-based predicates.

### 5.3. Evaluation Metrics

We evaluated the performance of FoXtrot by measuring both the time spent and the network traffic generated during indexing XPath queries and filtering XML data. We

Table I. Dataset Generation Parameters

| Parameter | Default | Range |
|---|---|---|
| Number of documents | $10^2$ | $10–10^3$ |
| Document depth | 10 | 5–25 |
| Number of queries | $10^6$ | $10^5–10^6$ |
| Query depth | 12 | 5–15 |
| Predicates per query | 2 | 1–3 |
| Wildcard probability | 0.2 | 0.2 |
| Desc. axis probability | 0.2 | 0.2 |

are also interested in how this traffic is distributed among the network peers. More formally, the metrics used in the experiments are the following. The *indexing latency* for a set of queries $Q$ is measured as the amount of time spent until all queries of $Q$ are indexed in the system. *Indexing throughput* is measured as the number of queries indexed over a specified time period. The *filtering latency* for a set of XML documents $D$ is measured as the amount of time spent until all notifications are dispatched to the interested subscribers for the queries matched by the documents of $D$.

The *network traffic* is measured as the total number of messages generated by network peers during indexing queries and filtering incoming XML data. We also distinguish the following types of peer load. First, the *filtering load* of a peer is measured as the total number of messages a peer sends during a filtering operation. Then, the *storage load* of a peer is measured as the total number of states it stores locally. Finally, we will use the term *NFA size* to refer to the total number of states included in the distributed NFA that is shared by the peers.

### 5.4. FoXtrot Setup

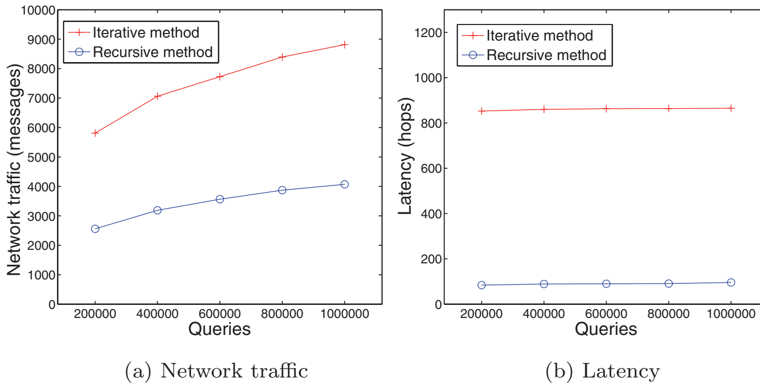
To carry out our experiments we execute the following steps. We create a network of $n$ peers connected using the Pastry DHT and implementing the functionality of FoXtrot. Then, we index a set of queries $Q$ in the system using randomly selected peers as subscribers and study the performance of FoXtrot with respect to the metrics just described. Finally, we filter a set of XML documents $D$, using random peers as publishers, and measure again all relevant metrics.

We have implemented only the recursive method in FoXtrot, since as expected and demonstrated through simulation results in our previous work [Miliaraki et al. 2008], it outperforms the iterative one in terms of latency because it distributes the load more evenly and generates less network traffic. Figure 5 presents two graphs from our previous work [Miliaraki et al. 2008] that illustrate this clearly.

In addition, we have designed a number of different techniques for value-matching. As shown in our recent study [Miliaraki and Koubarakis 2010], depending on which technique we employ, the generated network traffic varies, and different latencies are observed during indexing and filtering operations. Since we consider the comparison of these methods outside our scope, we only used the method, referred to as top-down evaluation with pruning, which has proven to perform best under various filtering scenarios. The default values for setting up FoXtrot, including the value of parameter $l$, are shown in Table II.

## 6. EXPERIMENTAL RESULTS

Our main goal during experimental evaluation is to demonstrate the scalability of FoXtrot and its load-balancing properties under various scenarios including a very large set of queries and a high rate of incoming data. The following experiments are divided into three groups. We begin our evaluation with load-balancing techniques for distributing the load imposed on the network peers aiming to improve the overall performance of FoXtrot. In the second group, we study the performance of FoXtrot during




(a) Network traffic (b) Latency

Fig. 5. Iterative vs. recursive method [Miliaraki et al. 2008]

Table II. FoXtrot Setup Parameters

| Parameter | Default |
|---|---|
| Network size (Cluster) | 112 |
| Network size (PlanetLab) | 396 |
| Structural matching | Recursive method |
| Value matching | Top-down with pruning |
| Parameter l | 2 |

query indexing. Then, in the third group of experiments, we demonstrate how FoXtrot operates during XML filtering. Finally, we summarize our evaluation by discussing our results. Unless otherwise stated, our results are obtained by running the experiments on the cluster. In cases where we observed differences among the experiments in the two environments, we point out these differences and discuss them in detail.

### 6.1. Load-Balancing

A core issue that arises in a distributed filtering system like FoXtrot is having peers sharing the load equally. This is important because if a fraction of peers becomes overloaded, the overall performance of the system can deteriorate. In systems like ONYX [Diao et al. 2004], it is required that a strategy be adopted for deciding where to store queries and how to deliver XML data using criteria like the topological distance between the broker and the data source, the available bandwidth, the content of the query, and the location of the subscriber. In ONYX, this selection process is performed by a centralized component. Instead, FoXtrot exploits the DHT mechanism and distributes a single NFA indexing all queries among the peers in a random way. This leads to a fairly uniform distribution of storage load among the network peers without requiring any additional action.

However, even when peers share the fragments of the NFA equally (i.e., storage load is evenly distributed), filtering load distribution can be very unbalanced. This is due to the tree-like structure of the NFA which causes peers responsible for the states with smaller depth to suffer more load than the others. Also, the distribution of element names in the XML document set being filtered can be skewed, causing the relevant states to be accessed more frequently. The same holds for the distribution of element names in the query set. Our main concern is balancing the filtering task that is the heaviest, consisting of both retrieving and executing the NFA states and dispatching notifications if a match is found.

In the following, we first describe the load-balancing methods we employ and then evaluate them experimentally. Note that we design our methods assuming a network

consisting of peers with similar capabilities and that our goal is to evenly distribute the load among them. We do not consider the case where the network consists of a heterogeneous group of peers.

*6.1.1. Static Replication.* Our state distribution technique assigns each state to one responsible peer. Increasing parameter $l$ can also affect the distribution of load, since peers are able to perform execution at a larger NFA fragment. However, this is not sufficient for achieving a uniform load distribution, and so we also exploit additional replication techniques.

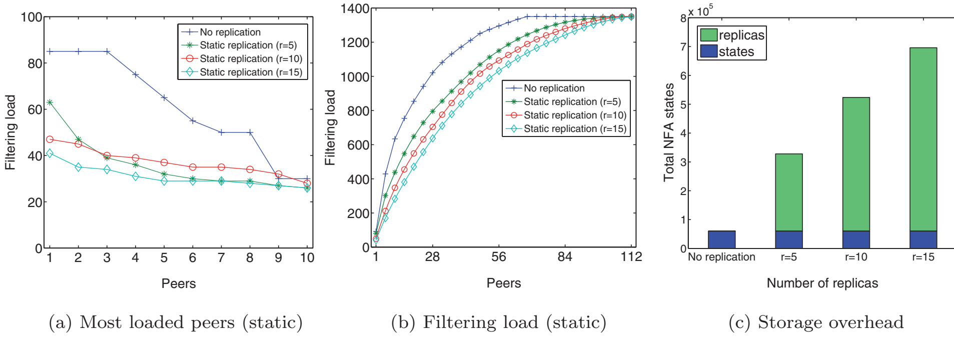
Our first method, called *static replication*, creates a fixed number of $r$ replicas, where $r$ is called the replication factor, for each NFA state. Replication takes place during query indexing and whenever a peer creates a state. This is accomplished by indexing each state $st$ with key $k$ using the additional keys $k_1 = k + 1, k_2 = k + 2, \ldots, k_r = k + r$, where operator $+$ denotes string concatenation. These correspond to the *replication keys* and lead to the peers responsible for the replicated states. During filtering, when a peer wants to forward a request for state $st$, it will choose randomly among the $r$ peers and the load that would be suffered by one peer is now distributed among the $r + 1$ peers. An obvious drawback of static replication is the extra storage overhead suffered by the peers as we increase the replication factor. Even if this overhead is considered negligible, it causes an increased latency during indexing, since $r$ times more states need to be created and stored. We demonstrate this in detail during our evaluation.

*6.1.2. Dynamic Replication.* To avoid the excessive storage requirements of static replication, which can cause latencies during indexing, we improve our method as follows. We assume that the frequency of visiting an NFA state during filtering is inversely proportional to the depth of this state. This assumption is made having in mind that the tree structure of the NFA is the main reason causing the load imbalances (e.g., if $r > 0$, one of the $r$ peers responsible for the start state will receive a filtering request each time an XML document arrives at the system). For this reason we create a different number of replicas for each state depending on its NFA depth. So, instead of having a fixed number of replicas for each NFA state, we create a number of $r/d$ replicas for each NFA state of depth $d$. We refer to this method as *dynamic replication*. Another interesting case is when the frequencies of visiting the NFA states are not dependent on the depth of the states but follow a different distribution. In this case, the number of the replicas for a state should be proportional to its access frequency $f$. Estimating these frequencies is an interesting problem, which we leave for future work.

*6.1.3. Evaluation.* In the following experiments, we evaluate our load-balancing methods using the following steps. We create a network of 112 peers, index $5 * 10^5$ path queries, and publish 100 XML documents simultaneously using random peers as publishers.

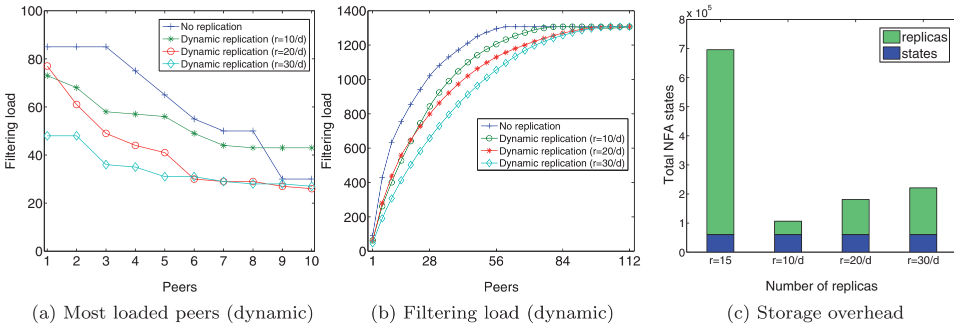
We begin with the evaluation of static replication while varying the number of replicas $r$ from 0 to 15. The results are presented in Figure 6. In Figure 6(a) we show the 10 peers that suffer the most load in a descending order of their filtering load. As we can see, when no replication is used, a fraction of peers is overloaded, receiving a large number of requests, while other peers receive only a small proportion of the total load. By adding even a small number of replicas in FoXtrot, load distribution is considerably improved. When 15 replicas are created, the 10 most loaded peers receive almost equal loads, eliminating potential bottlenecks.

In Figure 6(b) we show the overall load distribution. On the $x$-axis, peers are ranked starting from the peer with the most filtering load. The $y$-axis represents the cumulative filtering load, that is, each point $(x,y)$ in the graph represents the sum of filtering load $y$ for the $x$ most loaded peer. When no replication is used, the filtering load is very




(a) Most loaded peers (static) (b) Filtering load (static) (c) Storage overhead

Fig. 6. Load-balancing (I).




(a) Most loaded peers (dynamic) (b) Filtering load (dynamic) (c) Storage overhead

Fig. 7. Load-balancing (II).

unbalanced and many peers receive very few or no requests at all. In particular, more than 40 peers do not receive any filtering request from the total 1350 requests that are generated during filtering in FoXtrot (see the straight line segment when $x > 70$). By using replication, we quickly observe a more even distribution of load which improves as we increase the replication factor, and in addition all peers participate in the filtering process. We also measured the variation of the different peer loads using the metric of standard deviation ($\sigma$) and observed that deviation is decreased as we increase the number of replicas per state. For example, when no replication is used $\sigma \simeq 20$, while when $r = 15$, $\sigma \simeq 8$.

However, the price we pay for a more uniform distribution of the load is the large storage overhead suffered by the peers as we increase the total number of replicas. We are concerned with this mainly because it can delay indexing, since actual storage costs are negligible (measured in MBs). As shown in Figure 6(c), the number of replicas is high—to illustrate, when $r = 15$, the storage overhead is more than $6 * 10^5$ replica states. Note that storage load includes some redundant states due to parameter $l$, as discussed in Section 3.1. However, we do not create replicas for these states, and that is the reason a replica factor $r$ results in less than $r$ times the number of states.

We now continue with the evaluation of the dynamic replication method. We run the same experiments as before (the results are presented in Figure 7). We first demonstrate how load is distributed among the 10 peers that suffer most of the load. As we observe in Figure 7(a), as the replication factor is increased, the peer that receives the most filtering requests suffers less load. At the same time, the load is distributed in a more uniform way. Static and dynamic replication techniques exhibit a similar performance when $r = 15$ and $r = 30/d$, respectively (see also Figure 6(a)). The main

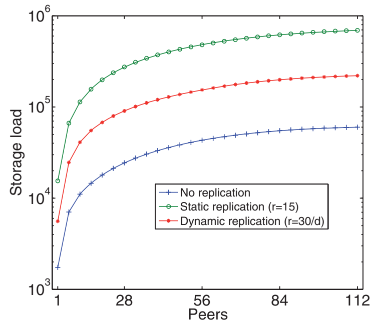



Fig. 8. Load-balancing (III): Storage load distribution.

advantage of dynamic replication is that we achieve this while keeping storage overhead low. As Figure 7(c) shows, a replication factor of $30/d$ almost triples the NFA states stored by the peers. This compares favorably with static replication, which achieves a similar load distribution for the case of 15 replicas (see Figure 6(b)), but the resulting amount of storage overhead is 9 times the number of states (see first bar of Figure 7(c)).

As previously, we also show the overall load distribution in Figure 7(b). Creating a varying number of replicas, depending on the depth of each NFA state, results in a more even distribution of load which improves as we increase the replication factor (from $r = 10/d$ to $r = 30/d$). When we use dynamic replication, all peers participate in the filtering process. We also measured the variation of the different peer loads using standard deviation and observed that deviation is relatively low. For instance, when $r = 30/d$, $\sigma \approx 10$ (the total number of filtering load is 1350 requests).

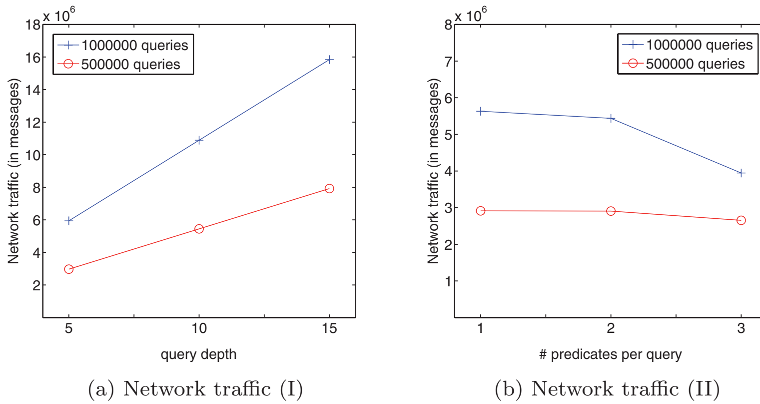
*Storage Load.* For completeness we also demonstrate in this group of experiments the storage load distribution in FoXtrot. The results are shown in Figure 8. We plot our results on a logarithmic scale because the total storage load differs considerably among the different load-balancing techniques. Again, the $y$-axis represents the cumulative load with peers ranked on the $x$-axis in descending order of their load. We can see in Figure 8 that even when no replication is used, as expected, storage load is distributed in a fairly uniform way due to the randomness of our distribution method. We report that a small group of peers stores a larger fraction of the total states, however in case of storage load, as we explained previously, these differences can be considered negligible (measured in MBs).

### 6.2. Indexing Queries

In this section we demonstrate how FoXtrot performs during query indexing. We are mainly interested in the number of messages that travel through the network and the time spent when indexing a set of queries.

*Network Traffic.* In this group of experiments, we study the network traffic that is generated during query indexing. We begin by examining the impact of query depth on the generated traffic and continue with how the number of predicates per query affects network traffic.

We create a network of 112 peers and index three different query sets containing queries with depths 5, 10, and 15, respectively. The results are shown in Figure 9(a) for the cases where $5 * 10^5$ and $10^6$ queries are indexed in FoXtrot. The graph shows the total amount of network traffic generated during the indexing of queries. In both cases, as Figure 9(a) depicts, the network traffic generated scales linearly with the depth of the queries being indexed. Particularly, for the case of $10^6$ indexed queries, as




(a) Network traffic (I)

(b) Network traffic (II)

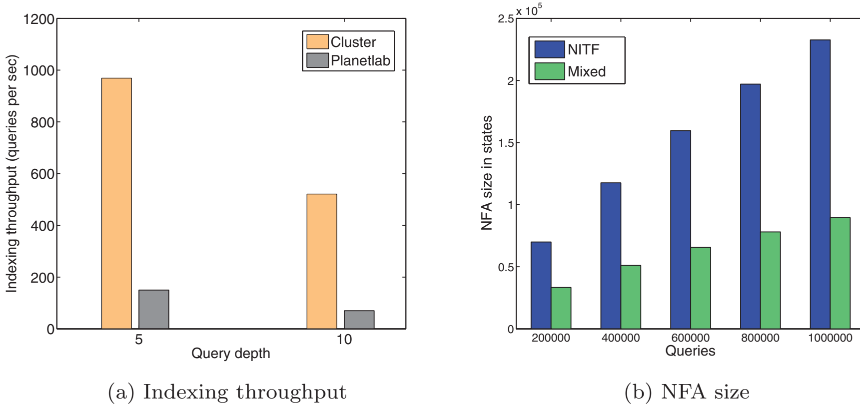
Fig. 9. Indexing operation (I).

we increase the query depth from 5 to 15, FoXtrot generates from $6 * 10^6$ to $16 * 10^6$ messages, respectively. This is due to the fact that indexing a single query of depth $d$ requires sending at most $d + 1$ messages, that is, one message to the peer responsible for the start state and $d$ additional messages to the peers responsible for the other $d$ states. These messages either update or create the corresponding NFA states. In some cases the messages actually sent may be slightly less, since peers can be responsible for subsequent states and less than $d + 1$ messages are needed for a query of depth $d$. Note that, for the purposes of this experiment, we prefer to index queries one at a time in each iteration. However, if queries arrive in chunks, we can decrease the number of messages by performing a bulk indexing operation for each chunk instead of several separate operations.

In Figure 9(b) we examine how the number of predicates per query affects network traffic during indexing query sets with 1, 2, and 3 predicates, respectively. We present two cases when $5 * 10^5$ and $10^6$ queries are indexed in FoXtrot. While network traffic increases linearly with the query depth, the total number of predicates included in each query does not significantly affect the number of indexing messages sent in all cases. We observe a decrease of network traffic as the number of predicates per query is increased. However, this is actually caused by the method we use for synthetically generating our queries. As we increase the number of predicates allowed per query, the query generator creates a set where queries share more structural similarities. In other words, the distributed NFA that is constructed is smaller and, as a result, fewer messages travel through the network during indexing. This is depicted more clearly in the case of $10^6$ queries, where we observe a 30% decrease on network traffic as we increase the predicates to 3 per query (average query depth in this case is 6).

*Indexing Throughput.* Next, we study the throughput of FoXtrot during query indexing measured as the number of queries indexed in a given amount of time. We report measurements for both the PlanetLab network and the cluster, since indexing throughput differs considerably in the two environments. The main reason is that network delays in a setting like PlanetLab, where peers are geographically dispersed, are significantly higher compared to the ones observed in the cluster.

Before proceeding with the results, we describe briefly a cache mechanism we used for decreasing latency. Since we repeatedly visit the same states of the distributed NFA by contacting the relevant peers, we cache useful routing information at each peer. Consider a peer $p$ that is responsible for a state $st$. Each time another peer $p'$ wants to forward an indexing message to $p$ as responsible for state $st$, the message will




(a) Indexing throughput (b) NFA size

Fig. 10. Indexing operation (II).

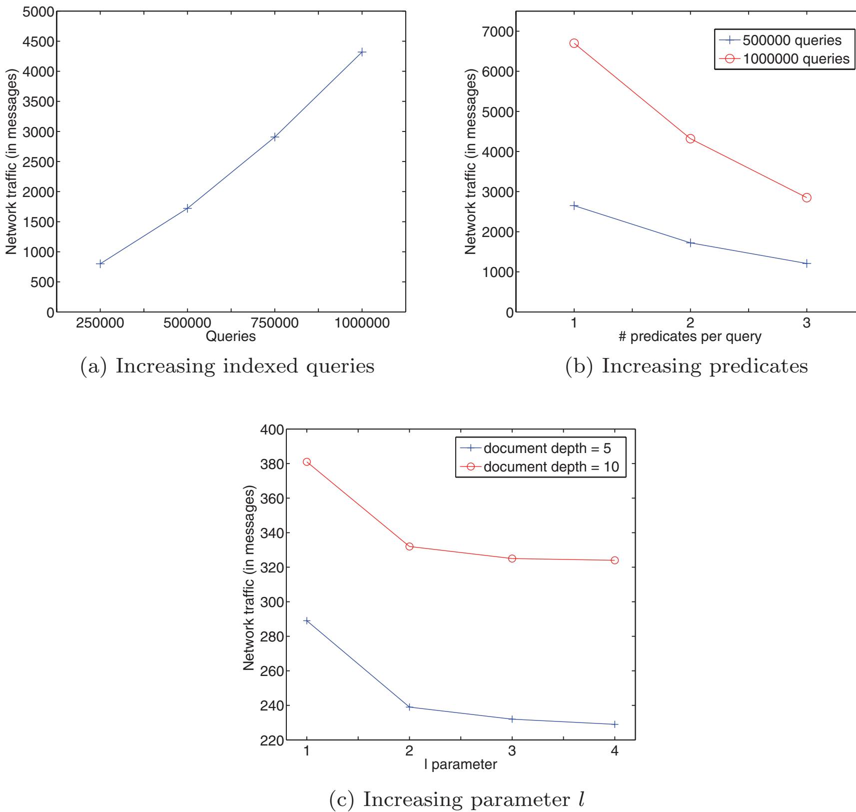
travel $O(\log n)$ hops to reach its destination.[1] We can avoid this by having $p'$ keep the IP address of $p$ as the peer responsible for state *st*. So if $p'$ wants to contact the peer responsible for state *st* again, it will first check its local cache and then the message will be delivered in a single hop. Such a caching technique is standard in these settings [Tryfonopoulos et al. 2005; Liarou et al. 2006] and helps reduce latency, since messages reach their destinations faster.

In Figure 10(a), we demonstrate the throughput achieved by FoXtrot in queries per second. In both cases, we create networks of 100 peers. In the case of PlanetLab, when query depth is 5, only 150 queries are indexed per second, while throughput drops to less than 70 queries per second when query depth is increased to 10 steps. This is due to the fact that as query depth increases, so does the indexing time, since the number of messages that are sent through the network are increased. FoXtrot exhibits a significantly better performance on the cluster, reaching a throughput of 969 queries per second when queries contain 5 steps.

We also report that in the case of PlanetLab, our measurements suffered from an increased variation. This was due to the existence of a few arbitrarily slow nodes. This problem has been studied in the context of a public DHT service, called OpenDHT, which was deployed on PlanetLab by Rhea et al. [2005]. The authors focused on the problem of slow nodes and demonstrated ways to overcome their effect on the performance of the system. We expect that we can further increase indexing throughput by performing a bulk indexing operation for each chunk of queries. This would benefit an environment like PlanetLab, where network latencies are large. Because of the strong dependence of indexing latency on the number of predicates per query, we do not include the results; the interested reader can refer to our recent study [Miliaraki and Koubarakis 2010].

*NFA Size.* We conclude our evaluation of query indexing with measurements for the NFA size shown in Figure 10(b) for both the mixed and the NITF datasets. We observe that the NFA size grows up to $2.5 * 10^5$ states when indexing $10^6$ queries from the NITF dataset, while in the case of the mixed dataset, the same number of queries is indexed in a much smaller NFA. In general, the number of the NFA states depends on the properties of the relative DTD and the characteristics of the query set. A larger number of elements allowed in a DTD results in a broader NFA (greater branching factor for each state), while a larger recursion level increases the depth of the

[1]Pastry routes messages to the peer whose identifier is numerically closest to the given key by using prefix routing. Each such request can be done in $O(\log n)$ steps, where $n$ is the number of nodes in the network.




(a) Increasing indexed queries

(b) Increasing predicates

(c) Increasing parameter $l$

Fig. 11. Filtering operation (I).

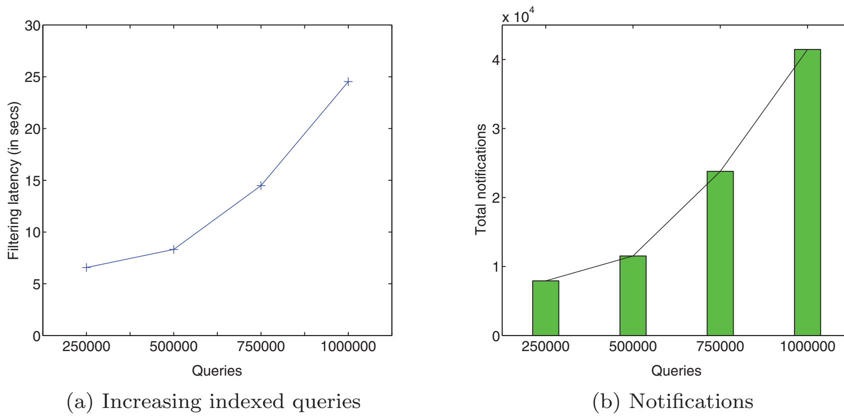
NFA. Note that the size of the distributed NFA differs from the size of the equivalent centralized NFA, since in FoXtrot we introduce redundancy by allowing peers to share and keep overlapping fragments of the NFA.

### 6.3. Filtering Documents

We continue our evaluation by studying the performance of FoXtrot during XML filtering. We are mainly interested in the number of messages that travel through the network and the time spent when filtering a set of XML documents.

*Network Traffic.* For the purposes of the experiments, we create a network of 112 peers and incrementally index $10^6$ path queries. After each indexing iteration, we publish, the whole document set consisting of 100 XML documents and measure the network traffic generated during the filtering of these documents. We repeat these steps for various cases. We do not consider the notification messages to be part of the network traffic.

First, we study how network traffic is affected as we increase the number of indexed queries and as a result we execute a larger NFA. The results are shown in Figure 11(a). We can see that network traffic scales linearly with the number of queries. As we index more queries in FoXtrot, the part of the distributed NFA that we traverse during filtering is larger, and as a result more messages travel through the network. We also examine how the number of predicates per query affects network traffic during filtering. In this case, we increase the number of predicates included in each query.




(a) Increasing indexed queries (b) Notifications

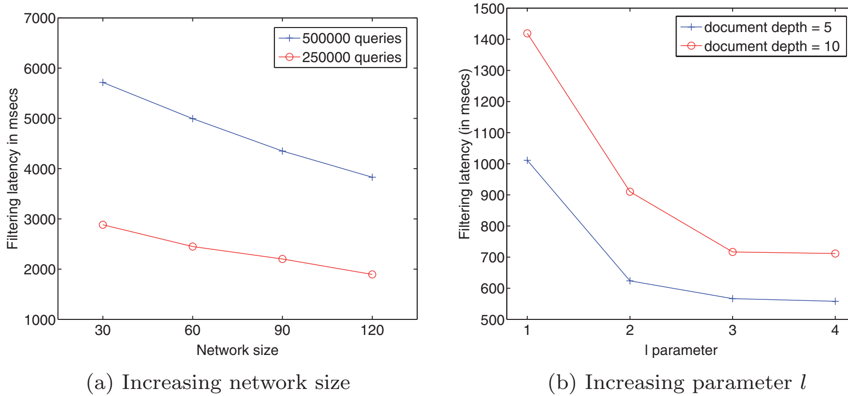
Fig. 12. Filtering operation (II).

Figure 11(b) shows the total amount of network traffic generated during filtering against the corresponding sets of queries with 1, 2, and 3 predicates. We present two cases when $5 * 10^5$ and $10^6$ queries are indexed, respectively. In both cases, the query set that contains more predicates is more selective, and this results in traversing a smaller part of the NFA during filtering. So network traffic is significantly decreased as the number of predicates per query increases, as we can see in Figure 11(b).

We continue by demonstrating how parameter $l$ affects the number of messages that are generated during filtering. The results are shown in Figure 11(c), where we measure network traffic as we increase $l$. We repeat our experiment for the cases where the average document depth is 5 and 10, respectively. We observe that increasing the value of parameter $l$ results in decreasing the generated amount of network traffic. This is because increasing $l$ enables each peer to perform execution on a larger path of the distributed NFA. However, we can see that as $l$ is increased, the corresponding decrease in the generated traffic is smaller.

*Filtering Latency and Throughput.* Apart from network traffic, we are also concerned with the filtering latency of the XML documents that arrive in FoXtrot. Recall that for a set of XML documents $D$, we measure filtering latency as the amount of time spent until all notifications are disseminated to the interested subscribers for the queries satisfied by the documents of $D$. As a result, filtering latency strongly depends on the number of notifications that are generated during filtering. We begin by studying how filtering latency is affected as we increase the total number of indexed queries (the results are shown in Figure 12(a)). As the graph shows, the time spent in filtering and delivering the notifications is proportional to the number of the queries matched in terms of the generated notifications. The number of these matches in each case is depicted in Figure 12(b). For example, $4 * 10^4$ notifications are generated when matching against $10^6$ queries (selectivity of 4%). In terms of throughput, when $10^6$ queries are indexed in FoXtrot, after publishing 100 XML documents, FoXtrot generates and disseminates about 1600 notifications per second.

We continue by demonstrating the scalability of FoXtrot during filtering as we increase the size of the network. We repeat our experiment for networks consisting of 30, 60, 90, and 120 peers accordingly. The results are shown in Figure 13(a) where two cases are depicted, when $2.5 * 10^5$ and $5 * 10^5$ queries are indexed, respectively, in the system. As the results clearly indicate when the size of the network increases, the time for filtering is significantly decreased. For example, when network size is increased




(a) Increasing network size (b) Increasing parameter $l$

Fig. 13. Filtering operation (III).

from 30 peers to 120 peers and $5 * 10^5$ queries are indexed in the system, filtering latency is decreased from 6 to less than 4 seconds.

We also study how we can improve filtering performance by increasing the value of parameter $l$. As shown in Figure 13(a), filtering latency is significantly decreased as we increase $l$, and this is mainly due to the smaller amount of network traffic that is generated (studied earlier and depicted in Figure 11(c)). We also observe that the margin for improvement is larger when the document set being filtered includes XML documents of a greater depth.

### 6.4. Discussion

Let us now summarize the results from our experimental evaluation. First, with respect to query indexing, FoXtrot is highly efficient, reaching a throughput of almost 1000 queries per second for a network of 112 peers deployed using the cluster machines. Even though we cannot directly compare FoXtrot to other systems evaluated under different conditions, we report on the performance of XNet [Chand and Felber 2008], which is a closely related system for distributed XML filtering. XNet is evaluated using an overlay consisting of 22 peers from the PlanetLab network. A total of $10^5$ queries are indexed in XNet, which exhibits an indexing throughput of almost 19 single-element (i.e., query depth is 1) queries per second for each peer.

We also studied the performance of FoXtrot during filtering by publishing a burst of 100 XML documents. FoXtrot exhibited a high filtering throughput, generating and delivering about 1500 notifications per second. Moreover, we demonstrated how scalable FoXtrot, since increasing its network size improved performance and decreased latencies. For load-balancing, we employed two simple yet effective replication methods for distributing the load among the FoXtrot peers. We demonstrated that using our dynamic replication method we can evenly distribute the filtering load while incurring a small storage overhead for the peers. We also illustrated how parameter $l$ affects the performance of FoXtrot and showed experimentally that increasing $l$ can help us decrease network traffic and improve filtering latency, especially for the case of deep XML documents. In general, depending on the specific properties of the distributed NFA and the size of the network, tuning parameter $l$ can lead to an improved performance.

The majority of these experiments were conducted in two different environments, namely the PlanetLab network and a shared cluster. PlanetLab represents the real-world conditions of the Internet, and for this reason we deployed FoXtrot on 396 nodes. The latency observed in PlanetLab, either during indexing or filtering, was always one

order of magnitude higher than the one observed in the cluster. We also experienced an increased variation in the measurements of Planetlab among our different runs.

## 7. RELATED WORK

In this section we survey related work. We begin our discussion with centralized XML filtering systems and continue with distributed ones. We give an extensive survey of distributed approaches, discuss their main properties, and refer to whether they deal with load-balancing in their settings. Finally, as FoXtrot is built on top of a DHT, we also survey related peer-to-peer systems.

### 7.1. Centralized XML Filtering

Many approaches have been proposed in the past for XML filtering in a centralized setting. One of the earlier ones was YFilter [Diao et al. 2003] and its predecessor XFilter [Altinel and Franklin 2000]. In the YFilter engine, an NFA is constructed from a set of XPath queries and is used as a matching engine that scans incoming XML documents and discovers matching queries. Other systems that also employ tree-based structures for XML filtering include XTrie [Chan et al. 2002], XPush [Gupta and Suciu 2003], and Index-Filter [Bruno et al. 2003]. A different approach is described by Tian et al. [2004] where they design and implement an XML publish/subscribe system using a relational database. As centralized solutions typically suffer from well-known disadvantages, including lack of scalability, creation of bottlenecks, and the existence of a single point of failure, it is more suitable to design a distributed system for offering XML filtering functionality on an Internet-scale. Thus we now survey such distributed approaches.

### 7.2. Distributed XML Filtering

The majority of distributed approaches [Snoeren et al. 2001; Chand and Felber 2003; Felber et al. 2003; Diao et al. 2004; Gong et al. 2005; Uchiyama et al. 2005; Chan and Ni 2007] assume an overlay network with content-based routers responsible for forwarding XML data towards interested subscribers. Content-based routers, also called brokers, route XML data based on their content, and are organized in mesh or tree-based configurations. We describe in detail the work that we consider most representative, emphasizing the network setting, the methods employed by the network brokers, and whether any load-balancing method is used for distributing the total load among the network brokers.

In the work of Felber et al. [2003], the authors first propose two simple strategies for parallelizing the filtering task, which is performed using the XTrie algorithm [Chan et al. 2002]. In the first strategy, called data-sharing, each router keeps the whole set of queries and a load balancer dispatches each XML document that arrives to one of the routers. In the second strategy, called query-sharing, routers share the queries equally and incoming XML documents are filtered by all the routers. Intuitively, the time for indexing queries in the former strategy is proportional to the number of routers in the network, while filtering time decreases as more routers become available. Likewise, filtering in the query-sharing strategy requires a broadcast operation to all routers. Apart from these strategies, the authors also propose to organize the XML routers in to a hierarchical structure and deal with the challenge of partitioning queries among the network brokers using the XTrie structure. In this approach, the authors acknowledge that certain routers may suffer heavy access load, but do not offer solutions to this problem.

In a more recent work by Chand and Felber [2008], the authors describe another XML content-based network called XNet. In XNet, filtering is also performed using the XTrie algorithm [Chan et al. 2002], and a global spanning tree is used to implement a broadcast layer for publishers to communicate with all the XML routers forming

the inner network. Chand and Felber [2008] focus on aggregation techniques for minimizing the size of the routing tables kept by the routers and employ fault-tolerance methods to recover from router failures. XNet is evaluated using an overlay consisting of 22 nodes from the PlanetLab network where 105 queries are indexed. They report on an indexing throughput of almost 19 single-element queries per second for each consumer node. With respect to load-balancing, the authors deal with the distribution of the routing load measured as the size of the routing tables.

Diao et al. [2004] present ONYX, assuming a similar topology to XNet, where each broker uses a broadcast tree for reaching all other brokers in the network. A centralized component, called the registration service, is used to assign *a priori* XML data sources and queries to brokers using criteria like topological distances between the source and broker, available bandwidth, query content, and the location of the subscriber. As a result, the registration service can suffer a lot of load. However, the authors do not address this issue or deal explicitly with the distribution of load among the network routers. In ONYX, the brokers' routing tables are instances of the YFilter engine [Diao et al. 2003] to enable forwarding messages only to those brokers that are interested in them. The authors also deal with message transformation, aiming to decrease incrementally the size of the XML messages that travel through the network.

SONNET [Zhou et al. 2007], is closely related to our work because it uses a DHT to build an XML dissemination system. In contrast to ONYX and XNet that forward XML data using queries as entries in the brokers' routing tables, Zhou et al. [2007] construct a summary of the queries using path digests. As a result, filtering is approximate and false positives are introduced. The authors use load-shedding techniques to balance the number of packets forwarded by each peer. However, they do not consider the size of each packet or the processing load suffered by each peer receiving a packet, and they evaluate their system through a simulated peer-to-peer network.

The XTreeNet system [Fenner et al. 2005] combines the publish/subscribe and the query/response models within a network of XML routers that connects XML data producers and consumers. The authors introduce the concept of content descriptors and create a different distribution tree for each of these descriptors. As a result, XML data are not matched repeatedly at internal brokers but travel through the different distribution trees. Content descriptors can be elements of an ontological topic hierarchy or XML data paths, and are considered high-level descriptions for both subscription and publication information.

A work that deals explicitly with load-balancing issues is described by Uchiyama et al. [2005], where the authors assume that they have available a number of servers that share XPath queries. Filtering is performed using a lazily constructed DFA similar to the work of Gupta and Suciu [2003]. The authors employ a load-shedding technique for transferring XPath queries from overloaded to under-loaded servers using a centralized component called the XPE control server.

Papaemmanouil and Cetintemel [2005] describe another relevant system, called SemCast, for distributed content-based routing. SemCast works with either relational or XML data, and in the case of XML, queries are expressed using the XPath language. In contrast to ONYX and other systems that require content-based filtering performed at all brokers, SemCast splits incoming data streams *a priori*, and sends them across multiple channels implemented as independent dissemination trees. The process of deciding which and how many channels are created is called *channelization*, and is performed in a centralized manner. This process can be revised periodically, and is based on criteria like network statistics, stream statistics, and profile characteristics. Similarly to XTreeNet [Fenner et al. 2005], the authors consider, as a key advantage for SemCast, that content-based filtering takes place only at the source and destination brokers, and do not focus on which filtering engine is used for this purpose.

Other work in the literature [Moro et al. 2007; Chan and Ni 2007; Gong et al. 2005] focus on optimizing the functionality of each single XML broker, and can be considered complementary to most of former work. For example, Gong et al. [2005] describe techniques that use Bloom filters for summarizing the queries in the routing tables.

### 7.3. Tree-Like Structures on Top of DHTs

We now describe approaches that distribute tree-like structures on top of DHTs and other publish/subscribe systems built using peer-to-peer networks. This work does not necessarily consider the XML data model; but since in FoXtrot we distribute an automaton on top of a DHT, we consider this work as closely related, and we discuss it in detail.

We begin with psiX [Rao and Moon 2009], a hierarchical distributed index for locating XML data in a DHT network. Each XML document and query is mapped into an algebraic signature, and indexes, called *H-indexes*, are built for the document signatures. To answer a query, first the root node of each H-index is discovered using the query elements. This is considered a special node, and its id is computed based on the relevant XML element name. XML data location continues with each peer following pointers to the other nodes that keep index entries. Note that apart from locating the root node, where a DHT lookup operation is used, a peer continues traversing an H-index by following a set of extra pointers kept locally. Apart from disk usage, the authors do not study how load is distributed as a result of their design, and depend solely on the underlying overlay offered by Chord for providing load-balancing of key-value pairs.

A similar approach to psiX was recently proposed by Abiteboul et al. [2008] with the system KadoP, which supports XPath query processing on top a DHT. Their indexing scheme is a combination of an inverted index on XML tags and a set of hierarchical indexes for storing the positional representation of tag name instances. Each peer keeps a list of indexes using B+-trees for XML tag elements. The authors acknowledge that distribution of element tags can be very skewed, and peers migrate their data in the case of popular terms.

Prior to the above work that refers to the XML data model, Aekaterinidis and Triantafillou [2006] designed PastryStrings, where DHT peers also keep additional pointers for traversing a forest of trees representing a set of queries. PastryStrings supports queries expressed using an attribute-value data model which handles a rich set of operators for both numerical and string attributes. Actually, they consider an alphabet $\beta$, and for each character of $\beta$, a tree structure is created (called a *string tree*) with words mapped to its tree nodes. Additional routing tables are kept by peers for enabling prefix-based routing. The authors are concerned with load-balancing since, as expected, a fraction of nodes—the tree nodes close to the root of each tree—may become bottlenecks. For this reason, they use common strategies, including replication of the trees, partitioning of the storage load for popular values, and also apply domain relocation techniques. The latter technique is based on the fact that each attribute is expected to have values from a very small part of its domain.

Zhang et al. [2005] propose a distributed tree scheme called Brushwood, designed on top of the Skip Graph DHT [Aspnes and Shah 2003], where peers are assigned a tree partition using a linearization of the tree. The authors target locality-sensitive applications like distributed file services. This work is concerned with load balancing and uses load shedding methods to achieve this. In particular, peer-wise gossiping is used to aggregate load information inside the distributed tree and then trigger load adjustment operations.

One of the earlier approaches that uses a trie structure for organizing data in a peer-to-peer system is the P-Grid system proposed by Aberer et al. [2003]. P-Grid uses

a virtual distributed search tree, structured similarly to DHTs for supporting both prefix and range queries. Queries are resolved using prefix matching, while the actual network topology has no hierarchy and each peer keeps a part of the overall trie. For reasons including fault-tolerance and load-balancing, multiple peers are responsible for each leaf node of the P-Grid trie.

A different approach is proposed by Jagadish et al. [2005], where the authors describe BATON (*BAlanced Tree Overlay Network*) for organizing peers in a distributed binary tree structure and supporting both exact match and range queries. In BATON, each peer stores tree nodes keeping links to its parent, children, adjacent nodes, and also some selected neighbors of the same level. For load-balancing, again a load-shedding technique is used where overloaded nodes share or migrate their data. However such a technique is not sufficient when global imbalances occur. BATON* [Jagadish et al. 2006] improves on BATON by supporting multiattribute queries using a multiway tree structure. This improved design allows achieving better load-balancing by increasing the fanout of the tree, leading to more leaf nodes. Also in this case, load-balancing occurs dynamically having peers partitioning or migrating their load when necessary. Other approaches include Prefix Hash Tree [Ramabhadran et al. 2004] that uses the lookup interface of a DHT to construct a trie-based structure for efficiently answering range queries.

### 7.4. Other Related Pub/Sub Systems

In the literature, there are many publish/subscribe systems that are based on data models and query languages different than ours. Such systems include SmartSeer [Kannan et al. 2006], Corona [Ramasubramanian et al. 2006], and the work of Tryfonopoulos et al. [2005]. Ramasubramanian et al. [2006] describe Corona, a publish/subscribe system built on top of a DHT. In Corona, each information source is assigned to a random peer which monitors the source and disseminates updates to interested clients who have subscribed to the specific source. The authors use an optimization method to decide which peers should monitor each channel using periodic polling aiming to optimize bandwidth utilization. In SmartSeer, the authors use a keyword-indexing method for allowing users to subscribe with queries containing conjunctions or disjunctions of terms over the CiteSeer database. Tryfonopoulos et al. [2005] design an information-filtering system supporting an attribute-value model in a DHT environment.

Apart from the many works that design publish/subscribe systems on top of structured overlays, other work like the system Sub-2-Sub proposed by Voulgaris et al. [2006] relies on gossip-based protocols and consider looser unstructured settings. The authors aim to support more complex subscription models, including interval-based subscriptions.

Finally, we point out that there are various interesting papers on storing XML documents in peer-to-peer networks and executing XPath queries [Bonifati et al. 2004; Galanis et al. 2003; Koloniari and Pitoura 2004; Skobeltsyn et al. 2005]. For example, Koloniari and Pitoura [2004] study content-based routing for XPath queries in a peer-to-peer network storing XML documents. Peers are connected using an unstructured peer-to-peer network and clustered based on their content. We do not present an in-depth discussion of these papers, since their emphasis is not on filtering algorithms.

## 8. CONCLUSIONS AND FUTURE WORK

We described FoXtrot, a fully distributed XML filtering system built on top of DHTs. FoXtrot combines the strength of an NFA for efficiently matching XPath queries and distributed hash tables for building a fully-distributed scalable system. We focus on

designing a load-balanced approach, avoiding bottlenecks that can deteriorate performance. Apart from structural matching performed using automata, we also discuss different methods for evaluating value-based predicates. We perform an extensive experimental evaluation of our system, FoXtrot, and demonstrate that it can index millions of user queries, achieving a high indexing and filtering throughput. At the same time, FoXtrot exhibits very good load-balancing properties, and is also scalable with respect to network size, since it improves its performance as we add more peers to the network. Our evaluation was done in a controlled environment of a local cluster and on the worldwide testbed provided by the PlanetLab network.

As subject of future work, we would like to consider extensions of the query language supported in this article, so that we reach full XPath. The first interesting extension that comes to mind is to consider branching paths. Such an extension would require a number of optimizations to ensure efficiency. For example, a query containing more than one linear path could be indexed using only its most selective path and the other paths would be checked only when a match is found. We also plan to study how our methods can be applied to richer data models, including the RDF data model [Manola and Miller 2004]. Since any path query can be transformed into a regular expression, and consequently there exists an NFA for representing this query, our techniques described using XML and XPath can be used for other data models and query languages (e.g., RDF path queries [Pérez et al. 2010]). Zhou and Wu [2010] also propose a different approach, decomposing RDF graphs to XML trees, and demonstrate improved query processing performance compared to existing RDF techniques. Such an approach can also be studied in the context of FoXtrot. With respect to our load-balancing methods, we should also consider the case where the frequencies of visiting the NFA states are not dependent on the depth of the states, but follow a different distribution. Estimating these frequencies and designing such a load-balancing method is an interesting problem, which we leave for future work.

## ACKNOWLEDGMENTS

We would like to thank Mema Roussopoulos for useful comments and discussions. We also thank Mihalis Nicolaou for implementing the initial algorithms for structural matching [Miliaraki et al. 2008].

## REFERENCES

ABERER, K., CUDR_E-MAUROUX, P., DATTA, A., DESPOTOVIC, Z., HAUSWIRTH, M., PUNCEVA, M., AND SCHMIDT, R. 2003. P-Grid: A self-organizing structured P2P system. *SIGMOD Record 32,* 3, 29–33.

ABITEBOUL, S., MANOLESCU, I., POLYZOTIS, N., PREDA, N., AND SUN, C. 2008. XML processing in DHT networks. In *Proceedings of the 24th IEEE International Conference on Data Engineering (ICDE'08)*. IEEE, Los Alamitos, CA, 606–615.

AEKATERINIDIS, I. AND TRIANTAFILLOU, P. 2006. PastryStrings: A comprehensive content-based publish/subscribe DHT network. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)*. IEEE, Los Alamitos, CA, 23–.

ALTINEL, M. AND FRANKLIN, M. J. 2000. Efficient filtering of XML documents for selective dissemination of information. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB'00)*. Morgan Kaufmann, San Francisco, CA, 53–64.

ASPNES, J. AND SHAH, G. 2003. Skip graphs. In *Proceedings of the14th Annual ACM-SIAM Symposium on Discrete algorithms (SODA'03)*. SIAM, Philadelphia, PA, 384–393.

BALAKRISHNAN, H., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. 2003. Looking up data in p2p systems. *Comm. ACM 46*, 43–48.

BARBOSA, D., MIGNET, L., AND VELTRI, P. 2006. Studying the XML Web: Gathering statistics from an XML sample. *World Wide Web 9,* 2, 187–212.

BONIFATI, A., MATRANGOLO, U., CUZZOCREA, A., AND JAIN, M. 2004. XPath lookup queries in P2P networks. In *Proceedings of the 6th Annual ACM International Workshop on Web Information and Data Management (WIDM'04)*. ACM, New York, 48–55.

BRUNO, N., GRAVANO, L., KOUDAS, N., AND SRIVASTAVA, D. 2003. Navigation- vs. index-based XML multiquery processing. In *Proceedings of the 19th International Conference on Data Engineering (ICDE'03)*. IEEE, Los Alamitos, CA, 139–150.

CHAN, C. Y., FELBER, P., GAROFALAKIS, M. N., AND RASTOGI, R. 2002. Efficient Filtering of XML documents with XPath expressions. In *Proceedings of the 18th International Conference on Data Engineering (ICDE'02)*. IEEE, Los Alamitos, CA, 235.

CHAN, C. Y. AND NI, Y. 2007. Efficient XML Data dissemination with piggybacking. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'07)*. ACM, New York, 737–748.

CHAND, R. AND FELBER, P. 2008. Scalable distribution of XML content with XNet. *IEEE Trans. Parallel Distrib. Syst. 19*, 4, 447–461.

CHAND, R. AND FELBER, P. A. 2003. A scalable protocol for content-based routing in overlay networks. In *Proceedings of the 2nd IEEE International Symposium on Network Computing and Applications (NCA'03)*. IEEE, Los Alamitos, CA, 123–.

CLARK, J. AND DEROSE, S. J. 1999. XML path language (XPath). Version 1.0. World Wide Web Consortium, Recommendation. http://www.w3.org/TR/xpath.

CONSENS, M. P. AND MILO, T. 1994. Optimizing queries on files. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'94)*. ACM, New York, 301–312.

DIAO, Y., ALTINEL, M., FRANKLIN, M. J., ZHANG, H., AND FISCHER, P. 2003. Path sharing and predicate evaluation for high-performance XML ffltering. *ACM Trans. Datab. Syst. 28,* 4, 467–516.

DIAO, Y., RIZVI, S., AND FRANKLIN, M. J. 2004. Towards an internet-scale XML dissemination service. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'04)*. VLDB Endowment, 612–623.

FELBER, P., CHAN, C., GAROFALAKIS, M., AND RASTOGI, R. 2003. Scalable filtering of XML data for Web services. *IEEE Internet Comput 7,* 1, 49–57.

FENNER, W., RABINOVICH, M., RAMAKRISHNAN, K. K., SRIVASTAVA, D., AND ZHANG, Y. 2005. XTreeNet: Scalable overlay networks for XML content dissemination and querying (synopsis). In *Proceedings of the 10th International Workshop on Web Content Caching and Distribution (WCW'05)*. IEEE, Los Alamitos, CA, 41–46.

FREEPASTRY RELEASE 2009. FreePastry 2.1 release. http://www.freepastry.org/FreePastry/.

GALANIS, L., WANG, Y., JEFFERY, S., AND DEWITT, D. J. 2003. Locating data sources in large distributed systems. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB'03)*. VLDB Endowment,874–885.

GONG, X., QIAN, W., YAN, Y., AND ZHOU, A. 2005. Bloom filter-based XML packets filtering for millions of path queries. In *Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*. IEEE, Los Alamitos, CA, 890–901.

GUPTA, A. K. AND SUCIU, D. 2003. Stream processing of XPath queries with predicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'03)*. ACM, New York, 419–430.

HOPCROFT, J. E., MOTWANI, R., ROTWANI, AND ULLMAN, J. D. 2000. *Introduction to Automata Theory, Languages and Computability*. Addison-Wesley, Boston, MA.

HOU, S. AND JACOBSEN, H. A. 2006. Predicate-based filtering of XPath expressions. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*. IEEE, Los Alamitos, CA, 53–.

IBM XML.1999. Generator 1999. IBM XML Generator. http://www.alphaworks.ibm.com/xmlgenerator.

JAGADISH, H. V., OOI, B. C., TAN, K., AND VU, Q. H. 2005. BATON:A balanced tree structure for peer-to-peer networks. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB'05)*. VLDB Endowment, 661–672.

JAGADISH, H. V., OOI, B. C., TAN, K., VU, Q. H., AND ZHANG, R. 2006. Speeding up search in peer-to-peer networks with a multi-way tree structure. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'06)*. ACM, New York, 1–12.

KANNAN, J., YANG, B., SHENKER, S., SHARMA, P., BANERJEE, S., BASU, S., AND JU LEE, S. 2006. Smartseer: Using a dht to process continuous queries over peer-to-peer networks. In *Proceedings of the IEEE INFOCOM*.

KOLONIARI, G. AND PITOURA, E. 2004. Content-based routing of path queries in peer-to-peer systems. In *Proceedings of the Advances in Database Technology (EDBT'04)*. Springer, 29–47.

LIAROU, E., IDREOS, S., AND KOUBARAKIS, M. 2006. Evaluating conjunctive triple pattern queries over large structured overlay networks. In *Proceedings of the International Semantic Web Conference*. 399–413.

LUA, E. K., CROWCROFT, J., PIAS, M., SHARMA, R., AND LIM, S. 2005. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Comm. Surv. Tutorials, 7,* 2, 72–93.

MANOLA, F. AND MILLER, E. 2004. RDF primer: W3c recommendation. Decision Support Systems.

MILIARAKI, I. 2011. Distributed filtering and dissemination of XML data in peer-to-peer systems. Ph.D. dissertations, Department of Informatics and Telecommunications, National and Kapodistrian, University of Athens.

MILIARAKI, I., KAOUDI, Z., AND KOUBARAKIS, M. 2008. XML data dissemination using automata on top of structured overlay networks. In *Proceedings of the 17th International World Wide Web Conference (WWW'08)*. ACM, New York, 865–874.

MILIARAKI, I. AND KOUBARAKIS, M. 2010. Distributed structural and value XML filtering. In *Proceedings of the 4th ACM International Conference on Distributed Event-Based Systems (DEBS'10)*. ACM, New York, 2–13.

MORO, M. M., BAKALOV, P., AND TSOTRAS, V. J. 2007. Early profile pruning on XML-aware publish/subscribe systems. In *Proceedings of the 33rd International Conference on Very large Data Bases (VLDB'07)*. VLDB Endowment, 866–877.

PAPAEMMANOUIL, O. AND CETINTEMEL, U. 2005. SemCast: Semantic multicast for content-based data dissemination. In *Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*. IEEE, Los Alamitos, CA, 242–253.

PEREZ, J., ARENAS, M., AND GUTIERREZ, C. 2010. nSPARQL: A navigational language for RDF. *Web Semant. 8*, 255–270.

RAMABHADRAN, S., RATNASAMY, S., HELLERSTEIN, J. M., AND SHENKER, S. 2004. Brief announcement: Prefix hash tree. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC'04)*. ACM, New York, 368–368.

RAMASUBRAMANIAN, V., PETERSON, R., AND SIRER, E. G. 2006. Corona: A high performance publish/subscribe system for the World Wide Web. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation (NSDI'06)*. Vol. 3. USENIX Association, Berkeley, CA, 2–2.

RAO, P. R. AND MOON, B. 2009. Locating XML documents in a peer-to-peer network using distributed hash tables. *IEEE Trans. Knowl. Data Eng. 21,* 12, 1737–1752.

RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. 2001. A scalable content addressable network. *SIGCOMM Comput. Commun. Rev. 31*, 161–172.

RHEA, S., CHUN, B.-G., KUBIATOWICZ, J., AND SHENKER, S. 2005. Fixing the embarrassing slowness of OpenDHT on PlanetLab. In *Proceedings of the 2nd Conference on Real, Large Distributed Systems (WORLDS'05)*. Vol. 2, USENIX Association, Berkeley, CA, 25–30.

ROWSTRON, A. AND DRUSCHEL, P. 2001. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed System Platforms (Middleware '01)*. Springer, Berlin, 329–350.

SKOBELTSYN, G., HAUSWIRTH, M., AND ABERER, K. 2005. Efficient processing of XPath queries with structured overlay networks. In *Proceedings of the OTM Conferences*. 1243–1260.

SNOEREN, A. C., CONLEY, K., AND GIFFORD, D. K. 2001. Mesh-based content routing using XML. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*. ACM, New York, 160–173.

STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'01)*. ACM, New York, 149–160.

TIAN, F., REINWALD, B., PIRAHESH, H., MAYR, T., AND MYLLYMAKI, J. 2004. Implementing a scalable XML publish/subscribe system using relational database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'04)*. ACM, New York, 479–490.

TRYFONOPOULOS, C., IDREOS, S., AND KOUBARAKIS, M. 2005. Publish/subscribe functionality in IR environments using structured overlay networks. In *Proceedings of the SIGIR*. 322–329.

UCHIYAMA, H., ONIZUKA, M., AND HONISHI, T. 2005. Distributed XML stream filtering system with high scalability. In *Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*. IEEE, Los Alamitos, CA, 968–977.

VOULGARIS, S., RIVIERE, E., KERMARREC, A.-M., AND VAN STEEN, M. 2006. Sub-2-sub: Self-organizing content-based publish/subscribe for dynamic large scale collaborative networks. In *Proceedings of the IPTPS*.

XMARK 2001. XMark: An XML benchmark project. http://www.xml-benchmark.org/.

YFILTER RELEASE. 2004. YFilter 1.0 release. http://yfilter.cs.umass.edu/code_release.htm.

ZHANG, C., KRISHNAMURTHY, A., AND WANG, R. Y. 2005. Brushwood: Distributed trees in peer-to-peer systems. In *Peer-to-Peer Systems IV, 4th International Workshop (IPTPS'05)*. Lecture Notes in Computer Science, Vol. 3640, Springer, Berlin, 47–57.

ZHOU, A., QIAN, W., GONG, X., AND ZHOU, M. 2007. Sonnet: An efficient distributed content-based dissemination broker (poster paper). In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'07)*. ACM, New York, 1094–1096.

ZHOU, M. AND WU, Y. 2010. XML-based RDF data management for efficient query processing. In *Proceedings of the 13th International Workshop on the Web and Databases (WebDB'10)*. ACM, New York, 3:1–3:6.

Received May 2011; revised March 2012; accepted May 2012