XML Data Dissemination using Automata on Top of Structured Overlay Networks

Iris Miliaraki[*]
iris@di.uoa.gr

Zoi Kaoudi
zoi@di.uoa.gr

Manolis Koubarakis
koubarak@di.uoa.gr

Dept. of Informatics and Telecommunications
National and Kapodistrian University of Athens
Athens, Greece

ABSTRACT

We present a novel approach for filtering XML documents using nondeterministic finite automata and distributed hash tables. Our approach differs architecturally from recent proposals that deal with distributed XML filtering; they assume an XML broker architecture, whereas our solution is built on top of distributed hash tables. The essence of our work is a distributed implementation of YFilter, a state-of-the-art automata-based XML filtering system on top of Chord. We experimentally evaluate our approach and demonstrate that our algorithms can scale to millions of XPath queries under various filtering scenarios, and also exhibit very good load balancing properties.

Categories and Subject Descriptors

H.3.4 [**Information Systems**]: Systems and Software—*Distributed Systems*; H.2.3 [**Information Systems**]: Languages—*Data description languages (DDL), query languages*; F.1.1 [**Theory of Computation**]: Models of Computation—*Automata (e.g., finite, push-down, resource-bounded)*

General Terms

Algorithms, Design, Experimentation

1. INTRODUCTION

Publish/subscribe systems have emerged in recent years as a promising paradigm for offering various popular notification services including news monitoring, e-commerce site monitoring, blog monitoring and web feed dissemination (RSS). Since XML is widely used as the standard format for data exchange on the Web, a lot of research has focused on designing efficient and scalable XML filtering systems.

In XML filtering systems, subscribers submit continuous queries expressed in XPath/XQuery to the system asking to be notified whenever the query is satisfied by incoming XML data. In recent years, many approaches have been presented for providing efficient filtering of XML data against large sets of continuous queries within a centralized server [5, 15, 10, 9, 23, 21, 25, 28]. However, in order to offer XML filtering functionality on Internet-scale and avoid the typical problems of centralized solutions, we need to deploy such a service in a distributed environment. Consequently, systems like [29, 12, 17, 16, 20, 32, 11] have employed distributed content-based routing protocols to disseminate XML data over a network of *XML brokers* or routers. XML brokers are organized in mesh or tree-based overlay networks [29, 16], and disseminate XML data using information stored in their routing tables. In the ONYX system [16], each broker has a broadcast tree for reaching all other brokers in the network, and also uses a routing table for forwarding messages only to brokers that are interested in the messages. The routing tables in ONYX are instances of the YFilter engine [15]. Other proposals like [20] employ summarization techniques like Bloom Filters for summarizing the queries included in the routing tables.

A major weakness of previous proposals for offering distributed content-based XML data dissemination is that they do not deal with how much processing load is imposed on each XML broker resulting in load imbalances. The processing load of a broker includes both filtering of incoming XML data, and delivering notifications to interested users. Unbalanced load can cause a performance deterioration as the size of the query set increases, resulting in a fraction of peers to become overloaded. In ONYX [16], a centralized component is responsible 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 dealing with load balancing. Other systems like [20] do not deal at all with the amount of load managed by each broker. However, load balancing in a distributed setting can be crucial for achieving scalability.

Another weakness of previous systems [29, 12, 17, 16, 20, 11] is that the size of the routing tables is dependent on the number of the indexed queries. Therefore, as the number of indexed queries increases, so does the size of the routing tables making their maintenance a bottleneck in the system.

In this paper, we deal with the aforementioned problems in XML dissemination systems and propose an alternative architecture that exploits the power of *distributed hash tables (DHTs)* (in our case, Chord [30]) to achieve XML filtering functionality on Internet-scale. DHTs have been used recently with much success to implement publish/subscribe systems for attribute-value data models [7, 22, 31]. We base our work on DHTs since we are interested in XML filtering systems that will run on large collections of loosely maintained, heterogeneous, unreliable machines spread through-

[*]This work was supported by Microsoft Research through its PhD Scholarship Programme.

out the Internet. One can imagine that collections of such machines may be used in the future to run non-commercial XML-based public filtering/alerting services for community systems like CiteSeer, blogs etc.

The main idea of our approach is to adopt the successful automata-based XML filtering engine YFilter [15] and study its implementation on top of a DHT. We show how to construct, maintain and execute a *nondeterministic finite automaton (NFA)* which encodes a set of XPath queries on top of a DHT. This *distributed* NFA is maintained by having peers being 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.

The main contributions of this paper are the following:

- We describe a set of DHT-based protocols for efficient filtering of XML data on very large sets of XPath queries. Our approach overcomes the weaknesses of typical content-based XML dissemination systems built on top of mesh or tree-based overlays. In particular, there is no need for a centralized component for assigning queries to network peers, since queries are distributed among the peers using the underlying DHT infrastructure (Section 4). Additionally, peers maintain routing tables of constant size regardless of the number of indexed queries in the system.

- We utilize the successful automata-based XML filtering engine YFilter [15] and study its implementation on top of a DHT. We present and evaluate two methods for executing the NFA (Section 5). In the iterative method a publisher peer is responsible for managing the execution of the NFA while states are retrieved from other network peers. The recursive method exploits the inherent parallelism of an NFA and executes several active paths of the NFA in parallel. The recursive approach preprocesses input XML documents and enriches them with positional information to achieve efficient parallel execution. Our focus is mainly on the recursive method which outperforms the iterative one in terms of network latency, while generating the same or even less network traffic.

- We show how to balance the storage load and the filtering load of the network peers without requiring any kind of centralized control (Section 6.4). To achieve this, we utilize two complementary techniques: virtual nodes [30] and load-shedding [19, 31].

- We present an experimental evaluation for studying the feasibility of our approach. We demonstrate that our techniques scale to millions of XPath queries for various workloads (Section 6).

2. BACKGROUND

In this section, we give a very 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.

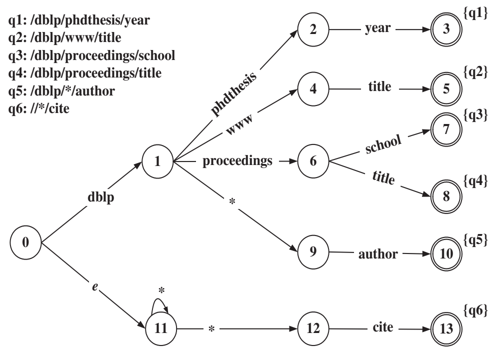


Figure 1: An example NFA constructed from a set of XPath queries

XPath [13] 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 zero or more predicates used to refine the selection of the node. We support predicates of the form $value_1$ *operator* $value_2$ where $value_1$ is an element name, an attribute name or the *position*() function, *operator* is one of the basic logical comparison operators $\{=, >, >=, <, <=, <>\}$, and $value_2$ is an element value, an attribute value or a number referring to the position of an element.

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 paper queries are written using this subset of XPath, and we will refer to such queries as *path queries* or *XPath queries* interchangeably. Queries containing branches can be managed by our algorithms by splitting them into a set of linear path queries. Example path queries for the DBLP XML database [1] are:

q1: `/dblp/phdthesis/[year=2007]`
which selects PhD theses published in year 2007.

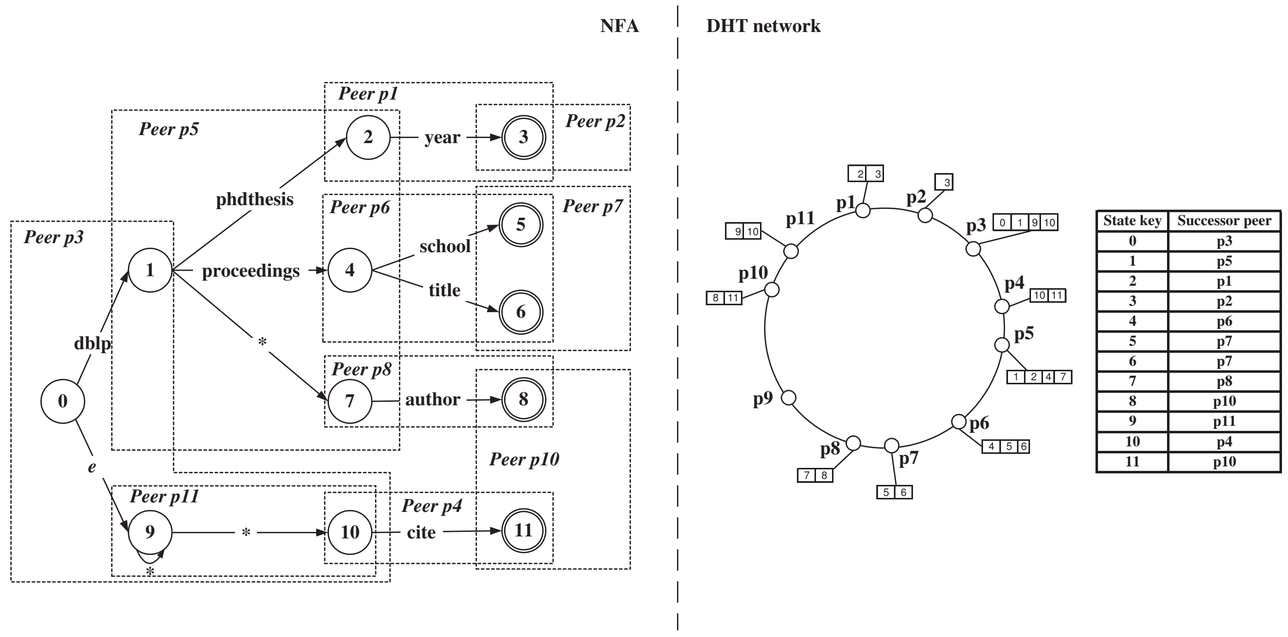
q2: `/dblp/*/author[@name="John Smith"]`
which selects any publication of author John Smith.

q3: `//*/[school="University of Athens"]`
which selects any publication written by authors from the University of Athens.

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, Σ 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 δ, 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 [24].

Any path query can be transformed into a regular expression and consequently there exists an NFA that accepts



State key	Successor peer
0	p3
1	p5
2	p1
3	p2
4	p6
5	p7
6	p7
7	p8
8	p10
9	p11
10	p4
11	p10

Figure 2: Distributing the NFA on top of a DHT network (l=1)

the language described by this query [24]. Following [15], for a given set of path queries, we will construct an NFA $A = (Q, \Sigma, \delta, q_0, F)$ where Σ 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 *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 δ. 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.

2.3 Distributed Hash Tables

DHTs like Chord [30] have emerged as a promising way of providing a highly efficient, scalable, robust and fault-tolerant infrastructure for the development of distributed applications. DHTs are structured P2P systems which try to solve the following *lookup* problem: given a data item x stored by a network of peers, find x. In Chord [30], each peer and each data item is assigned a unique m-bit identifier by using a hash function such as SHA-1. The identifier of a peer can be computed by hashing its IP address. For data items, we first have to select a key, and then hash this key to obtain an identifier for this data item. Identifiers are ordered on an identifier circle modulo 2^m, called the *Chord ring*. Key k with identifier *id* is assigned to the first peer whose identifier is equal to or follows *id* in the identifier space. This peer is called the successor peer of key k or identifier *id*, denoted by *succ*(k) or *succ*(*id*). Chord offers an operation `lookup(`*id*`)` that returns a pointer to the peer responsible for identifier *id*. When a Chord peer receives a `lookup` request for identifier *id*, it efficiently routes the request to *succ*(*id*) in $O(\log n)$ hops, where n is the number of peers in the network. This bound can be achieved because each peer in Chord maintains a routing table of size $O(\log n)$, called the *finger table.*

In the rest of the paper we use Chord as the underlying DHT. However, our techniques are DHT-agnostic; they can be implemented on any DHT that offers the standard lookup operation.

3. NFA-BASED DISTRIBUTED INDEX

We have selected to use an NFA-based model, similar to the one used in the system YFilter [15], for indexing queries in our system. The 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. In this section and Sections 4 and 5, we will describe in detail how the NFA corresponding to a set of XPath queries is constructed, maintained and executed on top of Chord.

The NFA corresponding to a set of path queries is essentially a tree structure that needs to be traversed both for inserting a query in the NFA, and for executing the NFA to find matches against an incoming XML document. We will distribute an NFA on top of Chord and provide an efficient way of supporting these two basic operations performed by a filtering system. 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. Moreover, an NFA was preferred to an equivalent DFA for reducing the number of states.

The states of the NFA are distributed by assigning 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 $\Sigma \cup \{\epsilon\}$, to a single peer in the network. Note that l is a parameter that determines how much 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 ϵ-transition, which are also stored at the peers responsible for

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

```
procedure $n$.IndexQuery($q$, $d$, $s$)
    $st$ is state at depth $d$ of $q$.NFA;
    if $n.states$ does not contain $st$ then add $st$ to $n.states$;
    // final state
    if $d = q.NFA.depth()$ then add $q$ to $st.queries$;
    else
        $t$ := transition label at depth $d$;
        if there is a transition labeled $t$ from $st$ to $st'$ then
            $nextPeer$ := Lookup($st'.key$);
            $nextPeer$.IndexQuery($q$, $d+1$, $s$);
        else
            add a transition labeled $t$ from $st$ to a new state $st'$;
            if ($t = \epsilon$) then
                $st'.selfchild$:=true;
            end
            $st'$.key := $st$.key + $t$;
            $nextPeer$ := Lookup($st'.key$);
            $nextPeer$.IndexQuery($q$, $d+1$, $s$);
        end
    end
end procedure
```

Procedure 2: `PublishDocument()`: *Publishing an XML document - Iterative way*

```
procedure $n$.PublishDocument($doc$)
    add $startState$ to $activeStates$
    $runtimeStack$.push($activeStates$);
    foreach $event$ from parsing $doc$ do
        if $event$ is a $startElement$ then
            foreach $st$ in $activeStates$ do
                add $st.queries$ to $satisfiedQueries$;
                $nextPeer$ := Lookup($st.key$);
                $states$ := $nextPeer$.ExpandState($st$, $event$, $myId$);
                add $states$ to $targetStates$;
            end
        else
            $runtimeStack$.pop();
        end
        $activeStates$ := $targetStates$;
        clear $targetStates$;
        $runtimeStack$.push($activeStates$);
    end
    foreach $q$ in $satisfiedQueries$ do
        notify subscriber of $q$;
    end
end procedure
```

the state which contains the ϵ-transition. 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 peers. Therefore, peers store overlapping fragments of the NFA and parameter l characterizes the size of these fragments.

Each state is uniquely identified by a key and this key is used for determining the peer that will be responsible for this state. The *responsible peer* for state with key k is the successor peer of $Hash(k)$, where $Hash()$ is the SHA-1 hash function used in Chord. 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*"+"*dblp*"+"*phdthesis*", the key of the start state is "*start*" and state 11 has key "*start*"+"\$", since ϵ-transitions are represented using character \$. Operator + is used to denote the concatenation of strings.

3.1 Peer local structures

Each peer p keeps a list, denoted by $p.states$, which contains the states assigned to p. Each state s included in $states$ is associated with a data structure containing the state's identifier, the transitions from this state, including potential self-loops and, in the case of accepting states, the identifiers and the subscribers of the relevant queries, in a list denoted by $s.queries$.

An example of how the NFA is distributed on top of Chord for $l = 1$ is depicted in Figure 2. We assume a network of 11 peers and each state stored is depicted on the Chord ring. Notice that state 10 is included in $p_3.states = \{0, 1, 9, 10\}$ because the ϵ-transition does not contribute to the specified length l. In the figure, we use unique integers instead of state keys for readability purposes.

4. CONSTRUCTING THE NFA

To achieve the above distribution of the NFA, the automaton is incrementally constructed as queries arrive in the system. We will describe first how YFilter constructs the NFA incrementally, and then describe how this has been adjusted in our work.

4.1 NFA construction in YFilter

The construction of the NFA in YFilter is done as follows. A location step can be represented by an NFA fragment [15]. 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 to *combine* the NFA of the query with the already existing one. 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 $L(R) \cup L(S)$. To insert a new query represented by an NFA S to an existing NFA R, we start from the common start state shared by R and S and we traverse R until either the accepting state of S is reached or we reach 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.

4.2 Distributed NFA construction

In our work, an NFA corresponding to a given set of path queries is constructed incrementally as queries are submitted, and it is distributed throughout the DHT. Thus, we will use the term *distributed NFA* to refer to it. We will now describe how a query q is inserted into the distributed NFA. The exact steps followed are depicted in Procedure 1. Algorithms in this paper are described using a notation, where $p.Proc()$ means that peer p receives a message and executes procedure $Proc()$.

Using Chord, the subscriber peer s sends a message `IndexQuery`(q, d, s) 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 and s is the subscriber peer. Initially $d = 0$ and r is the peer responsible for the start state, i.e. $succ$("*start*"). Starting from this peer, each peer p_1 that receives an `IndexQuery` message, checks the state st of q at depth d, retrieves the state with key $st.key$ from its local data structure $states$, and checks whether st is the accepting state of q. If so, p_1 inserts q in the list $st.queries$.

Procedure 3: `ExpandState()`: *Expanding a state by following transitions - Iterative way*

```
procedure n.ExpandState($st$, $event$, $publisherId$)
    $e$ := element name of $event$;
    foreach $element$ in $\{e, *, \epsilon\}$ do
        $st'$ := follow transition labeled $element$;
        if $st'$ is not null then
            add $st'$ to $targetStates$;
            if $element$ is $\epsilon$ then
                $nextPeer$ = LookUp($st'.key$);
                $nextPeer$.ExpandState ($st'$, $event$, $myId$);
            end
        end
    end
    if $st.selfState$ is true then add $st$ to $targetStates$;
    return $targetStates$;
end procedure
```

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

```
procedure $n$.RecExpandState($st$, $path$, $events$)
    if $st.isAcceptingState$ then
        add $st.queries$ to $satisfiedQueries$;
    end
    $event$ = $events$.next();
    if $event.hasSiblings()$ then
        $siblingEvents$ = $event$.getSiblings();
    else
        $siblingEvents$ = $\{event\}$;
    end
    foreach $e$ in $siblingEvents$ do
        if $e.isStartElement$ then
            compute $targetStates$ from $st$ for input $e$;
            foreach $s$ in $targetStates$ do
                $newPath$ = $path$.add($e$,$s$);
                $next$ = Lookup($s.key$);
                $next$.RecExpandState($s$, $newPath$, $events$);
            end
        end
        foreach $q$ in $satisfiedQueries$ do
            notify subscriber of $q$;
        end
    end
end procedure
```

Otherwise, let t be the label of the transition from state st to a target state st' as it appears in the NFA of q. If there is no such transition from st, p_1 adds a new transition to state st with label t with target state st'. Finally, p_1 sends a `IndexQuery`(q, d, s) message to peer p_2 with the same parameters except that depth d is increased by 1. If k is the key of state st, then $k + t$ is the key of state st' and p_2 is $succ(k + t)$.

For the sake of simplicity, Procedure 1 describes the base case for constructing the NFA when parameter l which determines the size of the NFA fragments assigned to each peer is 0. For larger values of l, each peer, instead of storing only the state that it is responsible for, it also stores a number of additional states defined by l. Constructing the NFA as described above, results in as many `IndexQuery` messages being sent to the network as the number of states in the NFA of q. Notice that the number of messages needed during the construction of the NFA is independent of the value of parameter l.

5. EXECUTING THE NFA

Again, we will first describe how YFilter operates for executing the NFA and then describe in detail our approach for executing a distributed NFA.

5.1 NFA execution in YFilter

The NFA execution proceeds in an event-driven fashion. The XML document is parsed using a SAX parser and the produced events are fed, one event at a time, to the NFA. The parser 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 *run-time stack*, while executing 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 element e, all active states are checked for transitions labeled with e, *wildcard* and ϵ-transitions. In case of an ϵ-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 run-time stack and become the active states for the next execution step. If a *EndOfElement* event occurs, the top of the run-time stack is popped and backtracking takes place. Execution proceeds in this way until the document has been completely parsed.

5.2 Distributed NFA execution

Let us now describe how we execute a distributed NFA in our approach. Similarly to YFilter, we maintain a stack in order to be able to backtrack during the execution of the NFA. For each active state, we want to retrieve all target states reached by a certain parsing event. Given that the NFA states in our approach are distributed among the peers of the network, at each step of the execution, the relevant parsing event should be forwarded to all the peers responsible for the active states. Therefore, we can identify two ways for executing the NFA: the first proceeds in an iterative way while the other executes the NFA in a recursive fashion.

In the *iterative method*, the publisher peer is responsible for parsing the document, maintaining the run-time stack and forwarding the parsing events to the responsible peers. In this case, the execution of the NFA proceeds in a similar way as in YFilter, with the exception that target states cannot be retrieved locally but need to be retrieved from other peers. Procedures 2 and 3 describe the actions required by the publisher peer and the actions required by each peer responsible for an active state.

The publisher peer p publishes a document by following the steps described in procedure `PublishDocument`(doc) where doc is the XML document being published. For each active state, p sends an `ExpandState`(st, $event$, $publisherId$) message to peer $r = succ(st.key)$, where st is the active state being expanded, $event$ is the current event produced by the parser and $publisherId$ is the identifier of the publisher peer. At first, the only active state is the start state.

When r receives a message `ExpandState`, it computes the transitions from state *st* that is stored in its local *states*, and returns to the publisher peer the set with all the target states computed. p continues the execution of the NFA until the document has been completely parsed. p is also responsible for notifying the subscribers for all the queries satisfied.

Although the iterative method performs poorly due to the fact that the majority of the load of the system is imposed on the publisher, we present it here for the reader's convenience (it might also be helpful in understanding the details of the recursive method). In the iterative approach, a stack mechanism is employed for maintaining multiple active paths during NFA execution. Each active path consists of a chain of states, starting from the start state and linking it with the reached target states. The main idea of the recursive method is that these active paths will be executed in parallel.

The details of the *recursive method* are 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 states result 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. Procedure 4 describes the actions required by each peer responsible for an active state during the execution of a path.

The publisher peer p publishes an XML document by sending a message `RecExpandState`(*st*, *path*, *events*) to peer $r = succ(\text{“}start\text{”})$, where *st* is the start state of the distributed NFA, *path* is a stack containing only the pair consisting of the event *StartOfDocument* and the start state, and *events* is the list with the parsing events. The parsing events *StartOfElement* and *EndOfElement* are enriched with a positional representation to efficiently check structural relationships between two elements. 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 in [14].

When peer r' which is responsible for state *st*, receives message `RecExpandState`, it first checks whether the next input event e in *events* has siblings. Note that e is determined by the last event included in stack *path*. If this is the case, then r' computes the transitions from state *st* for each sibling using its local data structure *states*. If st' is a target state reached from *st* then r' pushes the relevant sibling event along with st' onto the stack *path* and sends a message `RecExpandState`(st', *path*, *events*) to peer $r'' = succ(st'.key)$. 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. Then, r' will send $\sum \mid TS(e_i) \mid$ different messages, one for each of the different execution paths. The execution for each path continues until the document fragment has been completely parsed. Peers that participate in the execution process are responsible for notifying the subscribers of the satisfied queries.

Again, the steps described for executing the NFA, refer to the case where $l = 0$. For larger values of l, the operations `ExpandState` and `RecExpandState` instead of simulating the transition function δ, they simulate the extended transition function $\hat{\delta}$.

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 in [6]. However, in the case we want to filter larger XML documents, our method can be easily adjusted so that we forward smaller fragments of the document.

5.3 Predicate evaluation

We evaluate the predicates included in the queries after the execution of the NFA. This technique of delaying predicate processing until after the structure of a query is matched resembles the *Selection Postponed* (SP) approach presented in the original YFilter proposal [15]. An alternative approach called *Inline*, also presented in [15], evaluates the predicates of a query during the filtering process at each step of the NFA execution. As demonstrated in [15], SP outperforms Inline especially when queries contain a large number of predicates. This is mainly due to the fact that a lot of effort is spent evaluating predicates of queries which their structure may not be matched against the filtering data.

In future work, we plan to consider potential sharing of value-based predicates. Works like [23] and [25] identify the need for exploiting the potential commonalities between the predicates of the queries and not only structural commonalities.

6. EXPERIMENTAL EVALUATION

In this section, we present an experimental evaluation of our algorithms using a Chord [30] simulator we implemented in Java. All experiments were conducted on a machine with a Pentium IV 2.99 GHz processor and 4 GB memory running the Fedora Core 4 distribution of Linux. Our goal is to demonstrate the feasibility of our approach and study its overall performance for various workloads. The performance of our approach under churn or for different parameters characterizing the underlying DHT overlay is beyond the scope of this paper.

We generated two different document workloads to evaluate our approach. The first one is created using a set of 10 DTDs including DBLP DTD, NITF (News Industry Text Format) DTD, ebXML (Electronic Business using eXtensible Markup Language) DTD and the Auction DTD from the XMark benchmark [3] and will be referred to as *aggregated workload*. Using this workload, we study the performance of our approach in a realistic scenario where users subscribe to the same system to receive notifications concerning different interests of theirs (e.g., information about scientific papers vs. news feeds). The second workload is created using only the NITF DTD, which has also been used in [10, 15, 25], and will be referred to as *NITF workload*. We chose to run separate experiments using the NITF DTD because

Parameter	Aggregated	NITF
Number of documents	1000	1000
Dataset size (MBs)	6	1, 5
Number of elements	383	123
Number of attributes	694	513
Number of queries	10^6	10^6
Query depth	10	10
Query fanout	1	1
Wildcard prob.	0.2	0.2
Descendant axis prob.	0.2	0.2
Skewness of element names (θ)	0	0
Predicates per query	0	0
Network size	10^3	10^3

Table 1: Experiment parameters

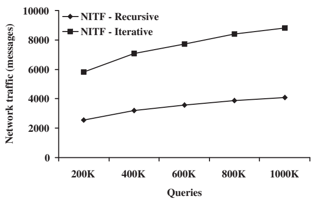
it represents an interesting case where a large fraction of elements are allowed to be recursive. For each workload 1000 documents are synthetically generated using the IBM XML generator [2]; 100 documents for each DTD in the aggregated workload. The path queries used in the experiments were generated using the XPath generator available in the YFilter release [4]. For each workload, we create two kinds of query sets: the *distinct* set which contains no duplicate queries and the *random* set which may contain duplicates. The random query set represents a more realistic scenario allowing subscribers to share interests.

The default values of the parameters used for generating the document workloads, the query sets and creating the network for our experiments are shown in Table 1. The number of the indexed queries does not affect the size of the routing table, which for a network of n peers is at most $\log n$ (in our case $\log 10^3$). Note that this is not true in XML content-based dissemination systems like [16], where the maintenance of large routing tables can become a bottleneck. The average document size is 4.8Kb in the aggregated workload and 36.5Kb in the NITF workload. Note that the average document size of an XML document in the Web is only 4 Kb, while the maximum size reaches 500Kb as mentioned in the study presented in [6]. In addition, these sizes are also typical for XML data dissemination settings [10, 11].

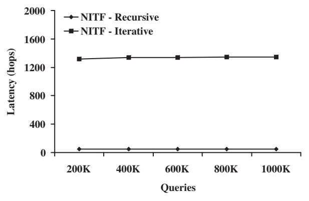
The metrics used in the experiments and their definitions are as follows. The *network traffic* is defined as the total number of messages generated by network peers while either indexing queries or filtering incoming XML documents. The *filtering latency* is measured in network hops as follows. While filtering an XML document, we measure the *longest chain of hops* needed during the execution of the NFA. In other words, we make the assumption that if a number of messages are sent simultaneously to a number of peers, the latency is equal to the maximum number of hops needed for a message to reach its destination. Lastly, we define the *NFA size* as the total number of states included in the NFA.

In Figure 4, we demonstrate the sizes of the NFAs for different DTDs. For instance, indexing 10^6 queries from the Auction DTD results in an NFA with 592199 states. Considering that in an Internet scale scenario we want to support millions of path queries from several DTDs, we can see the benefit of distributing such large NFAs to a large number of peers in a DHT network which cooperate for the NFA execution.

The following experiments are divided into four groups. The first group compares the iterative and the recursive method for executing the distributed NFA. The second group



(a) Network traffic



(b) Latency

Figure 3: Iterative vs. Recursive

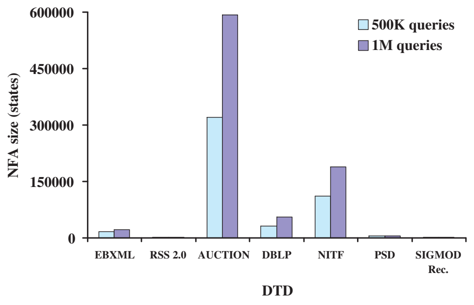


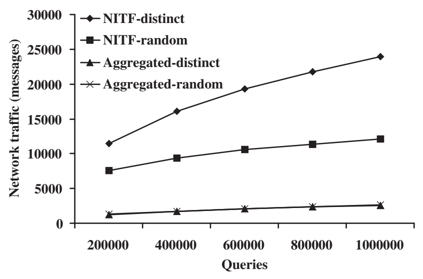
Figure 4: NFA size per DTD

studies the scalability of our approach for filtering XML data as the number of indexed queries increases. Then, in the third group of experiments we evaluate our approach as the size of the network increases. Finally, the fourth group demonstrates the effectiveness of load balancing techniques for distributing the load imposed on the network peers.

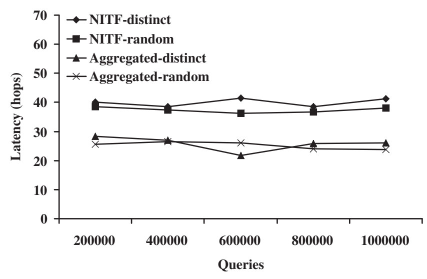
We have also evaluated our algorithms in the case where non-determinism in the query set increases (i.e., increasing probability of wildcards and descendant axis), and also for different values of l. The results of these experiments will appear in the extended version of this paper.

6.1 Recursive vs. Iterative

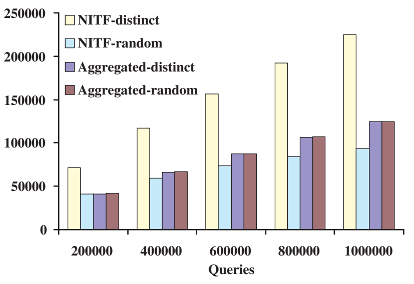
In this group of experiments, we compare the recursive and the iterative method for executing the distributed NFA. Even though the recursive method has obvious advantages over the iterative, we include the experiment for completeness. In this experiment, we created a network of 10^3 peers and incrementally indexed $2{*}10^5$ to 10^6 distinct path queries. After each indexing iteration, which doubles the amount of queries we have, we published 1000 XML documents and counted the average number of messages and the average filtering latency for filtering this set of documents. We show the results for the NITF workload in Figure 3. We observe that the iterative method needs approximately twice more messages than the recursive one, mainly due to response messages that are needed to be sent back to the publisher. In terms of filtering latency, the recursive method always outperforms the iterative method by more than 10 times. The reason for this is that the recursive method executes in parallel all active paths during the filtering of XML documents. Since the recursive approach always outperforms the iterative, in the rest of our experimental results, we only show the performance of the recursive method.



(a) Network traffic



(b) Latency



(c) NFA size

Figure 5: Varying number of queries

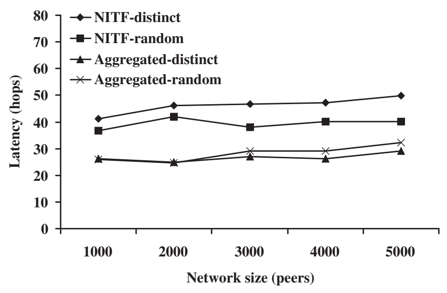


Figure 6: Varying network size (Latency)

6.2 Varying the number of queries

In this group of experiments, we study the performance of the system as the number of the indexed queries increases. We created a network of 10^3 peers and incrementally indexed $2 * 10^5$ to 10^6 distinct path queries. After each indexing iteration, which doubles the amount of queries we have, we published 1000 XML documents and counted the average number of messages and the average filtering latency for these documents. We run four different experiments: two for the aggregated workload and two for the NITF workload for both a random and a distinct query set. The results are shown in Figure 5.

As depicted in Figure 5(a), the generated network traffic for both workloads scales linearly with the number of queries. The NITF workload results in almost twice more messages than the aggregated one mainly due to the larger size of the constructed NFA. In Figure 5(c), the constructed NFA for the NITF workload consists of almost 225000 states while the corresponding size for the aggregated workload is 125000 states. Another factor that results in more network traffic for the random query set of NITF workload is its large document size.

In Figure 5(b), we show the filtering latency for both workloads. We observe that the latency remains relatively unaffected as the number of indexed queries grows. This happens because even though the size of the NFA increases, as more queries are added to it, the number of execution steps remains the same. In other words, the NFA reaches a state where the longest path corresponding to latency remains constant.

We omit the results for the generated network traffic and the latency during the indexing of queries in the network due to space limitations. Note that the network traffic generated during query indexing scales linearly with the length of the query automaton being indexed.

6.3 Varying the size of the network

This set of experiments evaluates how the performance of our approach is affected when we increase the size of the network. We created networks of 10^3 to $5 * 10^3$ peers and indexed 10^6 distinct path queries. After indexing the queries, we published 1000 XML documents and counted the average number of messages and the average filtering latency for filtering these documents.

We run two different experiments, one for the aggregated workload and one for the NITF workload. The results are shown in Figure 6. We observe that filtering latency remains unaffected as the size of the network increases for the same reasons explained above. We do not show graphs concerning network traffic as it remains unaffected as the size of the network increases. The reason for this is that the filtering process results in roughly the same number of messages regardless of the network size. We repeated the experiments for a random query set and we observed that the overall trends were similar.

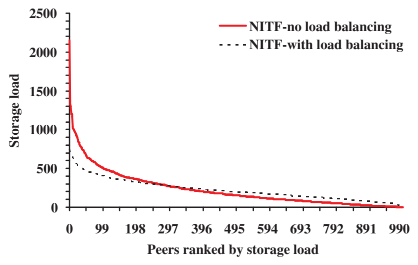
However the main motivation for using a larger network is to decrease the average load of each peer assuming that load is equally shared among the peers. The next section presents our techniques that allow us to ensure a balanced load in various settings.

6.4 Load balancing

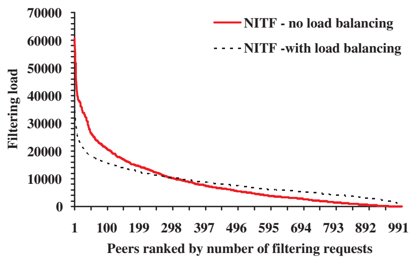
We distinguish between two types of load: *storage* and *filtering* load [31, 22]. The *storage load* of a peer is the total size of the states stored locally by the peer. The size of each state is defined as the number of transitions associated with it. The *filtering load* of a peer is the number of filtering requests that it processes (measured by the number of messages it receives during the filtering process).

For achieving a balanced storage load in our system, we can use a simple load balancing scheme proposed in Chord [30] which balances the number of keys per node by associating keys with *virtual nodes* and then mapping multiple virtual nodes (with unrelated identifiers) to each real peer. As demonstrated in [30], we need to allocate $logN$ randomly chosen virtual nodes to each peer for ensuring an equal partitioning of the identifier space between the peers.

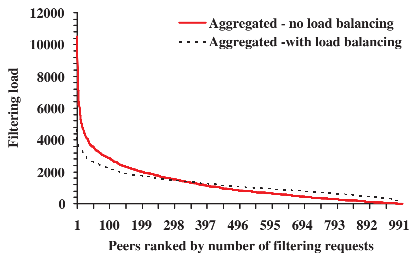
We run an experiment for demonstrating the effectiveness of the above load balancing technique for balancing the stor-



(a) Storage load (NITF)



(b) Filtering load (NITF)



(c) Filtering load (Aggregated)

Figure 7: Load balancing

age load in our approach. We created a network of 10^3 peers (each one assigned with 3 virtual nodes) and indexed 10^6 distinct path queries. Figures 7(a) shows the distribution of storage load across the peers. On the x-axis of this graph, peers are ranked starting from the peer with the highest load. For the NITF workload, we observe that prior to load balancing the first 200 peers received 118023 states which is 53% of total storage load and represents an unbalanced distribution. On the other hand, after assigning 3 virtual nodes to each network peer we achieve a fairer distribution of the storage load. Particularly, the first 200 peers received 86427 states or 38% of the total storage load.

Let us now consider the case where all peers share equal storage load. Then, if the states of the NFA are uniformly accessed, peers would equally share filtering load. However, the case where all states of the NFA are uniformly accessed is unrealistic. This is due to the inherent tree-structure of the NFA which results in skewness during accessing the states of the NFA. Peers responsible for states at smaller depths of the NFA will receive more filtering load than others. Furthermore, another factor that can cause additional imbalance during traversing the NFA is the potential skewness of elements contained in the XML document set being filtered. For this reason, using the *virtual nodes* load balancing scheme is insufficient for balancing the filtering load. For this reason, we have implemented and evaluated a load balancing method based on the concept of *load-shedding* used successfully in [19, 31]. The main idea is that when a peer p becomes overloaded, it chooses the most frequently accessed state st and contacts a number of peers requesting to replicate state st. Then, p notifies the rest of the peers that st has been replicated, so when a peer needs to retrieve it, it will randomly select one of the responsible peers.

To demonstrate the effectiveness of load-shedding, we run the following experiment. We created a network of 10^3 peers (each one assigned with 3 virtual nodes) and indexed 10^6 distinct path queries. A peer considers itself overloaded if more than 10% of the incoming documents access the same state that it is responsible for and then it assigns 10 peers to store a replica of that state. After each indexing iteration, which doubles the amount of queries we have, we publish 1000 XML documents and count the average filtering load suffered by each peer. We run the experiment for both aggregated and NITF workloads and the respective results are shown in Figures 7(b) and 7(c). We observe a significant improvement in load distribution when load balancing is used. For instance, prior to load balancing the first 50 peers receive almost 20% of the total filtering requests while the last 200 peers receive almost no load at all. In contrast, after applying the load balancing method, no peer receives more than 0.3% of the overall filtering load. We also experimented with different values for both the number of replicas and the access rate that determines if a peer is overloaded, but we did not observe any further improvements in the load distribution.

7. RELATED WORK

We classify related work in the area of XML filtering in two basic categories, centralized approaches and distributed approaches. Many approaches exist for XML filtering in a centralized setting: YFilter [15] and its predecessor XFilter [5], XTrie [10], XPush [23], the BEA streaming processor [18], the work of Green et al. [21], XSQ [28], Index-Filter [9] and others. As we already said in the introduction, little attention has been paid so far to providing a distributed Internet-scale XML filtering service. Systems like [29, 12, 17, 16, 20, 32, 11] have employed distributed content-based routing protocols to disseminate XML data over broker-based networks. XML brokers are organized in mesh or tree-based overlay networks [29, 16], and disseminate XML data using information stored in their routing tables. In the ONYX system [16], each broker has a broadcast tree for reaching all other brokers in the network and also uses a routing table for forwarding messages only to brokers that are interested in them. The routing tables in ONYX are instances of the YFilter engine. In [32], the authors concentrate on load balancing issues and present a system where XPath queries are transferred from overloaded to under-loaded servers by a centralized component called XPE control server. Other techniques like [20] employ summarization techniques like Bloom Filters for summarizing the queries included in the routing tables. Also, recent works like [27, 11] focus on optimizing the functionality of each of these XML brokers.

A recent system called SONNET [33], is most related to our work since it studies XML data dissemination on top of DHT. Each peer keeps a Bloom-filter-based engine for forwarding XML packets. Load balancing is concerned with balancing the number of packets forwarded by each peer regardless the size of each packet. Finally, [17] presents a hierarchical XML routing architecture based on XTrie. The authors describe both a *data-sharing* and a *filter-sharing* strategy for distributing the filtering task. The authors mention that they consider load balancing strategies but they do not provide any further details regarding a specific strategy.

Finally, we point out that there are various interesting papers on storing XML documents in P2P networks and executing XPath queries like [8, 19, 26]. We do not present an in-depth discussion of these papers since their emphasis is not on filtering algorithms.

8. CONCLUSIONS

We presented a novel XML filtering approach which distributes the state-of-the-art filtering engine YFilter [15] on top of structured overlays. Our approach differs architecturally from recent proposals [29, 12, 17, 16, 20, 32, 11] that deal with distributed XML filtering as these systems assume broker-based architectures, whereas our solution is built on top of DHTs. Our approach is the first that achieves load balancing between the network peers for XML data dissemination in a distributed setting without requiring any kind of centralized control. Additionally, peers keep routing tables of constant sizes whereas in systems like ONYX [16] the routing tables have size dependent on the number of the indexed queries causing it to become a bottleneck. We experimentally evaluate our approach and demonstrate that our algorithms can scale to millions of XPath queries under various filtering scenarios, and also exhibit very good load balancing properties. Our future work concentrates on implementing the proposed methods on a real DHT, evaluating them on a testbed like Planetlab and comparing them with other available systems.

9. REFERENCES

[1] DBLP XML records. http://dblp.uni-trier.de/xml/.

[2] IBM XML Generator. http://www.alphaworks.ibm.com/tech/xmlgenerator.

[3] XMark: An XML Benchmark Project. http://www.xml-benchmark.org/.

[4] YFilter 1.0 release. http://yfilter.cs.umass.edu/code_release.htm.

[5] M. Altinel and M. J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *VLDB 2000*.

[6] D. Barbosa, L. Mignet, and P. Veltri. Studying the XML Web: Gathering Statistics from an XML Sample. *World Wide Web*, 9(2):187–212, 2006.

[7] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting Scalable Multi-attribute Range Queries. In *SIGCOMM 2004*.

[8] A. Bonifati, U. Matrangolo, A. Cuzzocrea, and M. Jain. XPath Lookup Queries in P2P Networks. In *WIDM 2004*.

[9] N. Bruno, L. Gravano, N. Koudas, and D. Srivastava. Navigation- vs. Index-Based XML Multi-Query Processing. In *ICDE 2003*.

[10] C. Y. Chan, P. Felber, M. N. Garofalakis, and R. Rastogi. Efficient Filtering of XML Documents with XPath Expressions. In *ICDE 2002*.

[11] C. Y. Chan and Y. Ni. Efficient XML Data Dissemination with Piggybacking. In *SIGMOD 2007*.

[12] R. Chand and P. A. Felber. A Scalable Protocol for Content-Based Routing in Overlay Networks. In *NCA 2003*.

[13] J. Clark and S. J. DeRose. XML Path Language (XPath) Version 1.0. World Wide Web Consortium, Recommendation, November 1999.

[14] M. P. Consens and T. Milo. Optimizing Queries on Files. In *SIGMOD 1994*.

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

[16] Y. Diao, S. Rizvi, and M. J. Franklin. Towards an Internet-Scale XML Dissemination Service. In *VLDB 2004*.

[17] P. Felber, C.-Y. Chan, M. Garofalakis, and R. Rastogi. Scalable Filtering of XML Data for Web Services. *IEEE Internet Computing*, 7(1):49–57, 2003.

[18] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, J. Carey, and A. Sundararajan. The BEA Streaming XQuery Processor. *The VLDB Journal*, 13(3):294–315, 2004.

[19] L. Galanis, Y. Wang, S. Jeffery, and D. J. DeWitt. Locating Data Sources in Large Distributed Systems. In *VLDB 2003*.

[20] X. Gong, W. Qian, Y. Yan, and A. Zhou. Bloom Filter-Based XML Packets Filtering for Millions of Path Queries. In *ICDE 2005*.

[21] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing XML Streams with Deterministic Automata and Stream Indexes. *ACM Trans. Database Syst.*, 29(4):752–788, 2004.

[22] A. Gupta, O. D. Sahin, D. Agrawal, and A. E. Abbadi. Meghdoot: Content-based publish/subscribe over P2P networks. In *Middleware 2004*.

[23] A. K. Gupta and D. Suciu. Stream Processing of XPath Queries with Predicates. In *SIGMOD 2003*.

[24] J. E. Hopcroft, R. Motwani, Rotwani, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computability*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[25] S. Hou and H.-A. Jacobsen. Predicate-based Filtering of XPath Expressions. In *ICDE 2006*.

[26] G. Koloniari and E. Pitoura. Content-based Routing of Path Queries in Peer-to-Peer Systems. In *EDBT 2004*.

[27] M. M. Moro, P. Bakalov, and V. J. Tsotras. Early Profile Pruning on XML-aware Publish/Subscribe Systems. In *VLDB 2007*.

[28] F. Peng and S. S. Chawathe. XPath queries on streaming data. In *SIGMOD 2003*.

[29] A. C. Snoeren, K. Conley, and D. K. Gifford. Mesh-Based Content Routing using XML. *SOSP 2001*, 35(5):160–173, 2001.

[30] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *SIGCOMM 2001*.

[31] C. Tryfonopoulos, S. Idreos, and M. Koubarakis. Publish/Subscribe Functionality in IR Environments using Structured Overlay Networks. In *SIGIR 2005*.

[32] H. Uchiyama, M. Onizuka, and T. Honishi. Distributed XML Stream Filtering System with High Scalability. In *ICDE 2005*.

[33] A. Zhou, W. Qian, X. Gong, and M. Zhou. Sonnet: An Efficient Distributed Content-Based Dissemination Broker (Poster paper). In *SIGMOD 2007*.