Carleton University, Systems and Computer Engineering, Technical Report SCE-08-04, April 2008

Subscription Aggregation for Scalability and Efficiency in XPath/XML-Based Publish/Subscribe Systems

Abdulbaset Gaddah, Thomas Kunz

Department of Systems and Computer Engineering
Carleton University, 1125 Colonel By Drive
Ottawa, Ontario, Canada K1S 5B6
{agaddah, tkunz}@sce.carleton.ca

Ross Kouhi

Bell Laboratories, Service Infrastructure Research,
Alcatel-Lucent Canada Ltd., 600 March Road,
Ottawa Ontario Canada K2K 2E6
ross.kouhi@alcatel-lucent.com

Introduction

Content-based publish/subscribe systems provide a new communication paradigm to deliver relevant messages to various participants according to their expressed interests. In these systems, messages from senders (sources) are routed to receivers (ultimate destinations) based on their content, rather than a fixed destination address. Receivers describe their interests in receiving a particular category of messages by registering subscriptions, which are predicates on message content, in the system. When senders generate and inject messages in the network, the network routers (brokers) evaluate these messages against the registered subscriptions and route matched messages to their receivers. Content-based subscriptions provide receivers with a high degree of control over the type of information they wish to receive and hence limit the amount of network traffic. The challenging task of matching messages with many subscriptions is left to the network infrastructure, which is typically an overly network of distributed application-level routers.

Content-based routing has traditionally been used in the context of simple subscription languages, such as simple comparison predicates on attribute values, and the message is a simple dictionary data structure with name and value pair entries. As XML (eXtensible

Markup Language) [1] is widely used as a main standard format for interchanging a variety of data, there is increasing demands for XML-based pub/sub systems. Due to the flexible structure of XML documents, subscription specifications should be expressed by more expressive query languages such as XPath [2] or XQuery [3]. The recent use of XPath-based subscriptions to efficiently disseminate XML documents has increased the complexity of content-based routing. In this regard, we are interested in XML-based routing, where senders generate XML documents that are routed to interested receivers through application-level XML routers. Event receivers express their interest through XPath expressions.

A major key concern for content distribution networks is their scalability. As the number of XML documents and XPath-based subscriptions increases in the system, matching documents to subscriptions at line speed becomes a challenging problem [4, 5, 6]. Hence, there is an urgent need for optimization techniques to meet the performance challenges of routing XML documents. Typically, researchers propose to aggregate the subscriptions in either a loss-less [7] or lossy [8] form to reduce the filtering burden on the XML routers. These proposals represent the joint set of subscriptions (which are often a subset of XPath expressions) in a more compact data structure. As new subscribers join the system (or alternatively existing subscribers add new subscriptions), or existing subscribers leave or cancel existing subscriptions, these data structures need to be updated in some or all intermediate XML routers. These updates can be fairly complex, to ensure that previously aggregated subscriptions are now properly matching the new set of subscriptions.

This work will focus on a specific application domain for XML routing: disseminating sensor data. The input XML documents will describe sensor readings, similar to the work under development by the Open Geospatial Consortium [9, 10]. Most sensor readings will be encoded in relatively simple XML documents. Similarly, the XPath subscriptions will be, in general, linear expressions. We will research the most appropriate aggregation strategies for such a scenario, and explore the overheads in incrementally updating the necessary data structures in XML routers as

- New subscriptions are added.
- Existing subscriptions are removed

- The routing topology changes

XML documents and XPath queries

XML is a markup language that provides a widely adopted standard way of representing data in a flexible format. It allows us to define our own markup language and encodes the data of our documents in a more precise manner that can be easily processed. Typically, an XML document can be seen as a rooted, ordered, and labeled tree, where each node represents an element or a value and the edges correspond to (direct) element/subelement or element/value relationships. Each XML document has a single root node. There is a unique path from the root node to each element node in the document, which is referred to as *node path*. The depth of a node path is basically the number of nodes along that path. The maximal depth of all node paths is the XML document depth. For simplicity, this work focuses on the dissemination of XML documents which are small in depth.

XPath is one of the popular query languages that are proposed for XML data processing. The XPath path query can be viewed as sequences of location steps, where each node in the sequence is an element tag or a string value, and query nodes are related by either parent-child axes, indicated by a single line (/), or ancestor-descendant axes, indicated by a double line(//). In general, a simple path query of length l has the form "$a_1n_1a_2n_2 \ldots a_ln_l$", where each n_i is an element name or a wildcard symbol (*), and each a_i is either (/) or (//). For example, *"/A// * /D"* is a simple path query with length 3 that matches the following node paths: *"/A/B/D"*, *"/A/B/C/D"*, or *"/A/C/D"*. In this work, we consider simple liner path queries expressed by the above defined relations.

Existing XML Filtering Approaches and Challenges

We reviewed a number of existing approaches that focus on the scenario of XML data dissemination. XPath queries are pre-processed to create a routing table and a stream of XML documents are matched against the routing table entries for routing. There are basically two distinct approaches that address the problem of filtering and routing XML data: *Automaton-* and *index-based* approaches. In this section, we briefly describe the most popular filtering techniques that are based on these two approaches.

Automata-based Approaches: The state-of-the-art in XML data filtering includes Finite State Automaton-based (FSA) approaches such as XFilter [11], YFilter [12], DFA [13], AFilter [14], XScan [15], and XQRL [16]. In these approaches, some form of Finite State Machine (FSM) is adopted to represent XPath queries where path nodes of the queries are mapped to machine states. Each data node visited during parsing an incoming XML document triggers a state transition in the underlying FSM representation of the queries. A query is considered to match an input XML document when its final state is reached. The active states of the machine usually correspond to the prefix matches identified in the data. In fact, with a deep structure of XML documents, the number of active states can be exponentially large [12, 13, 17, 18]. As stated in [13], using an eager Deterministic Finite Automata (DFA) for a simple linear query may result in O(num_ancestor_axes × query_depth × alphabet_size × num_*_wildcards/num_ancestor_axes) active states. With multiple path queries, an eager DFA may have $O(2^{num\_path\_queries})$ active states. A lazy DFA is adopted to address the state explosion problem. Although the lazy DFA can sometimes be much smaller than the eager DFA, it is shown to be very memory intensive. The exponential state explosion can be clearly seen in XFilter as it builds a single FSM for each XPath query. This limits the scalability of XFilter to a small-scale filtering of XML data. In most approaches, the commonality among existing queries is not considered to avoid redundant processing of the queries. YFilter combines the input XPath queries into a single non-deterministic finite automata (NFA) structure to reduce the number of machine states. A run-time stack structure is used to maintain the active and previously visited states. However, since during runtime each NFA state can be visited (and inserted into run-time stack) several times, as indicated in [12], deep documents can practically cause an exponential explosion in the number of active, run-time states. AFilter consists of two data structures called AxisView (captures and clusters all axes of the registered queries in the form of a directed graph) and StackBranch (represent the current XML data branch). StackBranch is a compact stack-based structure that is used to traverse the AxisView structure to identify if there are any matches in the current data branch when trigger events occurred. Each time a start tag is encountered in the data stream, a stack object is created and pushed into its corresponding stack. The StackBranch structure still needs to store an exponential number of objects for deep

documents, which may increase the memory requirement. A hybrid XML filtering engine [36], a combined structure of DFA and AFilter, introduced to filter simple and complex queries separately. Simpler XPath queries, ones without wildcards (*) or descendents (//), are stored in a combined DFA while AFilter structure is used to store more complex path queries. The hybrid structure supports subscription insertion/deletion in an incremental process. A single DFA and index-based structures are built in parallel to hold simple and complex queries respectively. Whenever a new subscription query is received as part of insertion request, it has first to be parsed to identify if it is a simple or complex query. If the query is a simple one, it is simply add to the DFA structure by traversing DFA until either the final state is reached, or no edge is found for some location step in the query. In the later case, the remaining location steps of this query are added to the DFA. If it is a complex query, then the indexed structure is checked to determine the accepting (final) state of the new query. If it is present, the structure nodes will be traversed to add new labels of the location paths of the query in reverse order. The query is successfully added to the structure when the root node is reached. If an element is not found during any step in the process, a new node must be created and added to the structure to represent that element. The process of query deletion is performed in the similar way. Before the adding and deleting operations, queries are stored in temporary buffer until they are successfully committed. It is not clear the reason behind using two different structures to deal with simple and complex queries separately. Although DFA structure can provide an efficient mean to process XPath queries, it is expected to perform poorly with a large number of XPath queries, because the structure size grows exponentially with size of the workload.

Index-based Approaches: Various index-based approaches were proposed to match path queries against XML documents. This includes, but is not limited to, XTrie [19], PathStack/TwigStack [20], FiST [21], Index-Filter [22], and PathM [23]. In general, index-based techniques combine path queries into a prefix tree and generate an element position index for the incoming XML data. Then, the prefix tree is computed based on the index for the matched queries. XTrie is an index structure that offers an efficient way of filtering XML documents based on XPath queries. It represents XPath queries as strings and indexes them into a trie-based data structure, called XTrie, which leverages prefix commonalities in filters. XTrie supports several features that make it attractive for

content-based XML routing. First, it provides an effective filtering engine for matching a large number of complex, tree-structured XPath queries (as opposed to simple, single-path queries) against XML data. Second, XTrie supports both ordered and unordered matching of XML documents. Third, by using indexing technique in a trie-based structure along with a sophisticated matching algorithm, XTrie can efficiently minimize the number of unnecessary index probes and avoid redundant matching, thereby speeding up the filtering process. PathStack/TwigStack introduced two families of index-based path/twig join algorithms as primitives for matching path queries against an XML document efficiently. Here, twig queries are typically a subset of XPath expressions that include parent/child, ancestor/descendent axes, and node predicates. The proposed algorithms are generalizations of the binary structural join algorithms introduced in [20, 23, 24] to match path and twig queries. Their technique mainly depends on the use of a chain of linked stacks to compactly represent partial results to query paths, which are then stitched together to obtain matches for the twig pattern. The core contribution of the PathStack/TwigStack algorithms is that no large intermediate results are generated for complex path or twig queries, thereby eliminating the need for an optimization step that was needed when composing partial results from the algorithms in [23, 24]. In particular, the TwigStack algorithm showed to be I/O and CPU efficient for a large set of query twig patterns. FiST (Filtering by Sequencing Twigs) converts twig queries expressed in XPath and XML documents into sequences. These sequences are organized into a dynamic index-based data structure for efficient filtering. Instead of matching individual linear paths and then performing post-processing to identify matching twig queries, FiST exploits *holistic* matching of twig queries with incoming XML documents. The matching is holistic since the twig query is matched as whole rather than matching individual linear paths from root-to-leaf. FiST supports holistic matching by transforming twig queries and incoming documents into Prufer [21] sequences with inherently support for ordered query matching. Unlike XTries, since these sequences represent each filter query holistically, each query pattern is filtered independently without leveraging any prefix sharing. Index-Filter is a novel technique to answer multiple path queries by using indexes to build structural information over the tags in the XML document. By taking advantage of this additional information, Index-Filter is able to avoid processing large portions of the input

document that are guaranteed not to be part of any match. It also generalizes the PathStack algorithm, and takes advantage of a prefix tree representation of the set of path queries to share computation during multiple query evaluation. PathM uses a compact data structure to encode pattern matches rather than recording them explicitly as several XPath streaming algorithms [26, 27] do when both predicates and descendant axes are present in the path queries, and the XML data is recursive (i.e. data in which tags are repeated along a root-to-leaf path). Explicitly storing pattern matches by enumeration can be expensive in terms of memory size. PathM also uses a polynomial time streaming algorithm to evaluate a large set of XPath queries over streaming XML data. The algorithm searches for satisfying matches by probing the compact data structure in a lazy manner without enumerating all the pattern matches.

Key Optimization Techniques

Recently, XPath-based subscriptions are used to express the interest of consumers in receiving certain XML data. In large-scale content-based systems, matching a large volume of such subscriptions against XML documents at line speed becomes a challenging issue. Several optimization techniques are proposed to meet the performance challenges of content-based filtering and routing. Two key optimizations are considered to reduce the matching burden on the XML routers. The first optimization, which has gained much attention, uses indexing techniques ([5, 12, 13, 19, 21, 22, 29, 30, 31]) to perform selective matching with only a compact subset of potentially matching subscriptions. The second optimization uses aggregation techniques to convert an initial set of subscriptions into a compact and generalized subset of subscriptions to minimize the matching overhead [4, 6, 7, 8, 33, 34, 35]. This section reviews the most popular optimization techniques that were found in the literature.

Bloom Filter [5] introduced a novel technique for XML data filtering. A Bloom filter is basically a bit-vector of length m used to efficiently represent XML path queries of one user. Initially, all the vector bits are set to 0. Then, a number k of independent hash functions are selected to map the user queries into its Bloom filter. This results in setting some vector bits to 1's. To determine the existence of a match, the bits of a user Bloom filter are checked using the same hash functions. If any of them is set to 0, this will imply

a matching failure. In contrast, if all of them are set to 1, this suggests that a match is found (with some probability of a false positive, i.e., the Bloom filter mistakenly indicates a match while it is not). It is shown in [28] that the probability of a false positive is negligible and acceptable by most applications. Figure 1 shows an example of a Bloom filter with 4 hash functions.

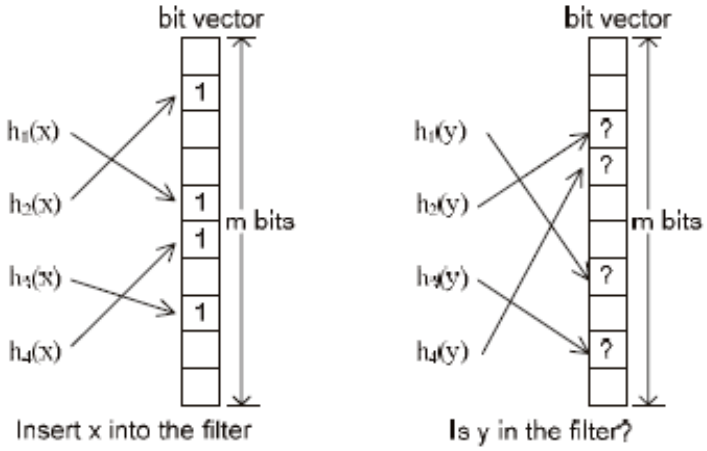


Figure 1: A simple Bloom filter with 4 hash functions [5]

In general, the routing table consists of many Bloom filters that are representations of the XML path queries. During the parsing process of each incoming XML document, a set of candidate paths is generated. Each candidate path is mapped to a bit-vector by the same hash functions to be evaluated against the routing table entities. If the existence of a candidate path is observed in a user's Bloom filter, the related XML data is forwarded to the user. It is obvious that the number of candidate paths increases exponentially with the depth and the hierarchal structure of an XML document. Thus, it becomes the bottleneck of the system. To improve the filtering performance, a new data structure, *Prefix Filters*, is introduced to decrease the number of candidate paths. Figure 2 provides an example of prefix filters. . For each path query string, there are different query prefixes. For example, query string "/A//B/*" has respectively length 2 and 3 prefixes "/A//B" and "/A//B/*". Prefix filter L_i is a Bloom filter representing length i prefixes of all users' queries. During parsing an XML document, each new created candidate path of length l will be matched against prefix filter L_l. If it does not match the prefix filter, none of all users' queries will match it. The system will discard this candidate path.

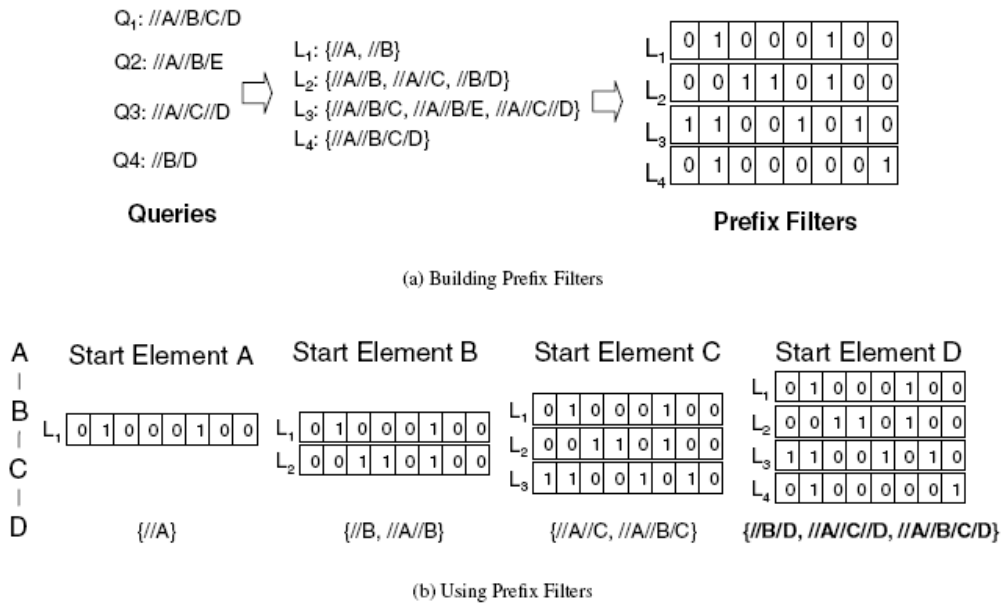


Figure 2: An example of prefix filters [5]

The idea of a Counting Bloom Filters is used to avoid any conflict in the bit vector that may occur due to deleting an XML path query. A single counter is assigned to each bit in the Bloom filter to track the number of hashed items related to that bit. Whenever an item is inserted to or removed from the Bloom filter, the counters related to the k hash values are increased or decreased, respectively. As a counter value changes from zero to one or from one to zero, correspondingly the related bit in the Bloom filter is set to 1 or 0.

It is worth mentioning that Bloom filter is an incremental process, i.e., path queries can easily be added to or deleted from the routing table. This is considered as a key benefit of the Bloom filter-based approach. The authors have only focused on transforming path queries to a compact representation to store large number of queries in an efficient manner. They have not considered the similarity among user queries, which could lead to redundant transformation process and increase the building time and size of routing table. The performance of the Bloom filter can be significantly affected when the depth of the input XML documents is large. This is mainly due to the large number of generated candidate paths.

The authors of [29] proposed a general approach to index a large number of subscriptions to quickly discover and maintain covering relationship. Their indexing technique is

dynamic, in which subscriptions can be added to and removed from the database. The key benefit of detecting covering relationship among subscriptions is to significantly reduce routing table size and avoid unnecessary proliferation of subscriptions throughout the system. Whenever a new subscription S arrives, the index is used to discover if there is a current subscription covering S. If this is the case, subscription S will not be forwarded. The index only examines a small fraction of subscriptions stored in the database to identify covering subscriptions. When the subscription S is deleted, some subscriptions previously covered by S may no longer be covered by any other subscription. Hence, such subscriptions need to be identified and routed to other routers. To facilitate this task, a data structure, called *relation graph*, is used to maintain already discovered covering relationship among subscriptions. The proposed solution is basically structured in two layers. The relation graph represents the upper layer, which stores the already discovered covering relationship among subscriptions. The actual index represents the lower layer, which is used to discover new covering relationship during the subscription or unsubscription process. Figure 3 shows an example of a relation graph containing 5 subscriptions numbered according to their arrival order.

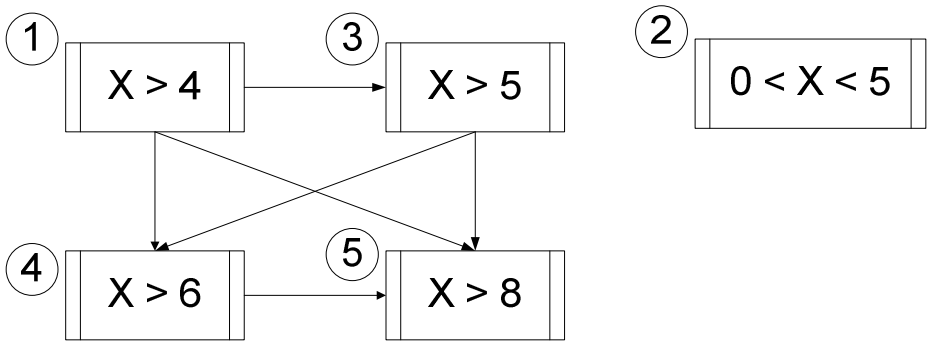


Figure 3: An example of a relation graph [29]

The relation graph is not easy to maintain due to its covering redundancy. For example, if a new added subscription S is covered by all existing subscriptions, then edges need to be built from each existing subscription into S. This can be an expensive process, especially if S is purged immediately. The cost of adding new subscriptions increases proportionally with the number of coverings. Similarly, deleting a subscription S that covers other subscriptions requires rebuilding the edges of the covered subscriptions. As a result,

Subscription insertion and deletion is not performed in an incremental manner.

The work by [32] proposed a novel approach to optimize the performance of an XML router by reducing the overhead of subscription matching. Their approach, which is called *piggyback* optimization, enables a downstream router to leverage the subscription matching work performed by upstream routers to reduce its own filtering overhead. This kind of collaboration is achieved by piggybacking some additional useful information in the form of header annotations in the XML documents being routed to downstream routers. When an XML document arrives, an XML router first pre-processes the header annotations to optimize subsequent processing of the XML document. The annotated information helps in making any immediate routing decisions, or reducing the effective number of subscriptions that need to be matched. Figure 4 shows the aggregated subscriptions in (a), the XML document in (b), and the routing tables in (c). In Figure 4(c), R_i is used to denote a router, T_i to denote the routing table, $A_{i,j}$ to denote annotated information, and D to denote the XML document.

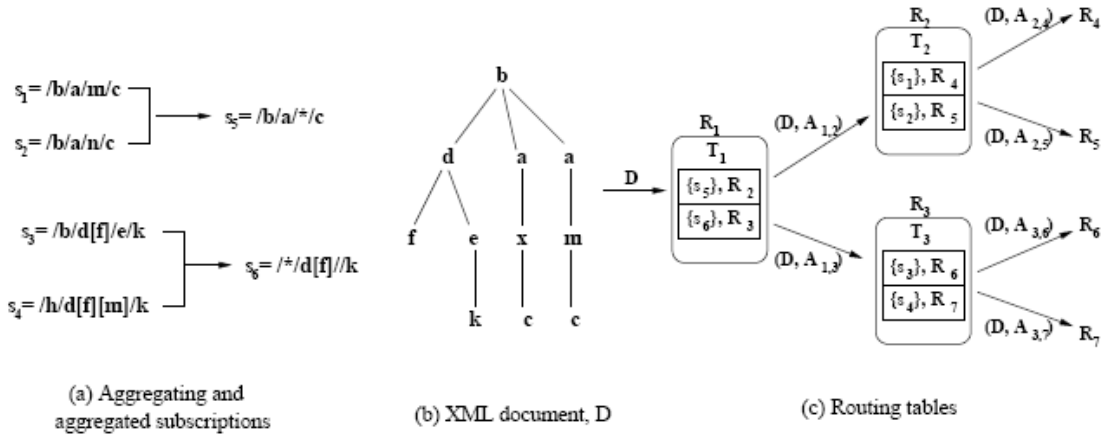


Figure 4: XPath subscriptions, XML document, and routing tables [32]

During the matching process in an upstream router, some useful information is acquired about an XML document and how the matched subscriptions are related to it. Such knowledge is classified into positive and negative information (annotations). Positive annotations correspond to the information related to either (1) subscriptions in the routing table that matched the XML document or (2) data patterns observed in the document. Negative annotations correspond to the information related to either (1) subscriptions in

the routing table that did not match the XML document or (2) data patterns that did not occur in the document. The annotations are created in two steps, referred to as the offline and online steps. The former is performed only once as part of the routing protocol to set up the routing tables in the routers. To generate more effective annotations, some useful information related to the subscriptions in the downstream routers is exploited. More precisely, a downstream router transmits such information to each of its upstream routers when it advertises its aggregated subscriptions. The upstream routers will locally store this information and use them to generate annotations in the online step for the documents that are routed to the downstream routers. The later step is achieved by an upstream router each time it requires to route a document to some downstream routers.

With a large population of subscriptions and XML documents, the computation overhead incurred by the upstream router to build annotations can offset any performance gains of its downstream routers. Also, the performance of downstream routers can be degraded as they are required to transmit additional information to the upstream routers and process incoming annotations. It should be noted that larger annotations can add additional overhead in terms of parsing and transmitting them. Redundant annotation can be created for similar subscriptions stored in the routing table. The probability of *false positives* can arise which are acceptable and do not compromise correctness. Although subscriptions can be incrementally added and deleted, the updating process may affect the accuracy of the information that is piggybacked on the incoming XML documents. This is because there is a strong relation between the subscriptions and the piggybacked information.

XRoute [4, 7, 33] proposed a content-based routing protocol for XML-based data dissemination systems. To optimize network traffic and bandwidth, the XRoute protocol ensures perfect routing (i.e., an XML document is delivered only to those consumers that have submitted a matching subscription). Moreover, it takes full advantage of subscription aggregation to minimize the size of the routing tables and the processing time at the XML routers. Subscriptions aggregation is a key technique to support a large number of subscriptions. Subscriptions can be dynamically registered and cancelled without affecting the routing accuracy. The XRoute protocol implements two forms of subscription aggregation. If subscription S_1 and S_2 arrive through the same interface to a

node (as the case in N_3 shown in Figure 5), it is said that S_2 is represented by S_1 at that interface. Here, it is assumed that S_1 covers S_2 (i.e., any event matching S_2 also matches S_1). In contrast, if both subscriptions arrive through different interfaces to a node (as the case in N_1 shown in Figure 5), it is said that S_2 is substituted by S_1. The aggregation mechanism is derived from the following observation: when an event e is received by node N_3, it is only necessary to examine e against S_1 due to the covering property. Thus, subscription S_2 becomes redundant and should not be propagated upstream from N_1 to N_3. Instead, S_2 can be aggregated with S_1 and only S_1 is forwarded to upstream node N_3.

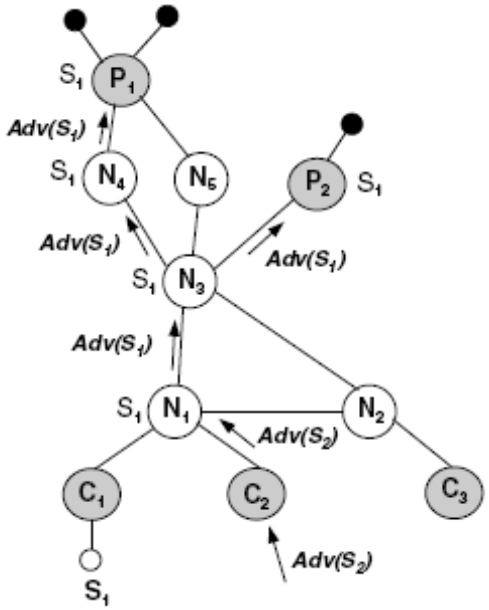


Figure 5: A sample content-based pub/sub network [4, 7, 33]

Each subscription entry, in the routing table, maintains some information about all the registrations of subscription S that is received by node N. Such information represents N's view of its neighbors whom they are interested in subscription S. Also, the subscription entry includes additional information required to implement the aggregation mechanism. This information is used to build the aggregation relations between subscription S and the other subscriptions in the routing table. Establishing the aggregation relations is an expensive process as it requires modifying all existing relations. The authors have not clarified how their aggregation mechanisms deal with subscription cancellation. However, we suspect that cancellation process is not performed in an incremental way due to the required update in the covering relations and substitute

pointers. The substitution mechanism adds some complexity to the routing protocol as it is expected to guarantee perfect routing and consistent system state during subscription cancellation. Redundant subscriptions are eliminated during the building of the covering relations. It is claimed that the matching accuracy is free of false positives or negatives.

The authors of [6] presented their experience in using an advertisement-based technique for optimizing data dissemination in a content-based system. In general, advertisements are announcements to the information that a data producer will generate in the future. They are used to limit the propagation of subscriptions only to the producers who advertise what the consumers are interested in. In their approach, the advertisements are generated by using XML Document Type Definition (DTD) information to define the legal building blocks (root to the leave) of related XML documents. Along with the advertisement-based subscription routing, optimization techniques, such as covering and merging, are proposed to identify the covering relations among XPath queries and to merge similar XPath quires. They mainly aim to reduce the routing table size stored at each router and speed up the routing process.

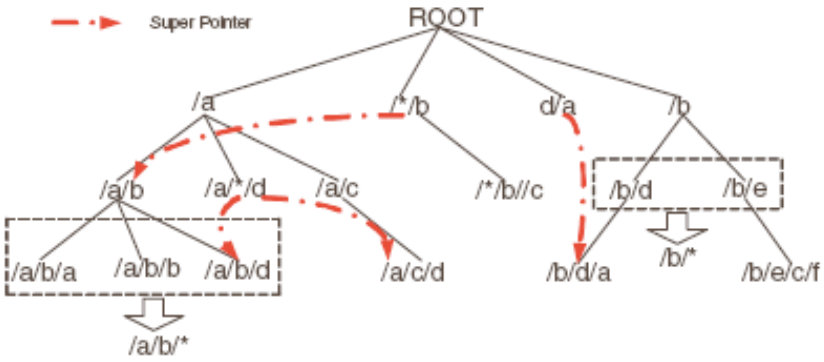


Figure 6: Subscription tree [6]

A data structure, called *subscription tree* shown in Figure 6, is exploited to capture the covering relations among registered subscriptions. The basic concept of covering is described as the following: when an incoming subscription is covered by a current subscription in the routing table, this subscription will not be routed to the neighboring brokers. In contrast, when the new subscription covers current subscriptions, before it is routed, the router needs to unsubscribe all the current subscriptions (the covered ones). This results in eliminating redundant subscriptions in the routing tables. Subscriptions are

stored in the subscription tree according to their covering relationships. Each node in the tree represents a single subscription that covers all subscriptions in its sub-tree. Although a subscription node can have only one parent in the tree, it may be covered by several subscriptions. To achieve this, a set of super pointers is used by each node to indicate the covering relations with subscription nodes outside its sub-tree. In the absence of covering relations among a set of subscriptions, subscription merging can be used to build a more compact routing table. In the subscription tree, siblings of the same parent node are better candidates to be merged. As indicated in Figure 6, node /a/b/a, /a/b/b, and /a/b/d can be merged into a new subscription node /a/b/*, which is the union of the three original expressions. As shown in Figure 6, When two nodes, /b/d and /b/e, are merged to /b/*, their children become the new node's children.

In the advertisement-based technique, as the publisher needs to update its advertisements frequently, the network traffic as well as the overhead of matching the generated advertisements at each router can be drastically increased. This may hamper the gain of reducing subscription broadcasting among the routers and add to the already large subscription matching costs. In a large subscription tree, identifying the covering relationships among subscriptions can be costly due to sequential search. Some techniques, like subscriptions indexing [5, 12, 21], should be supported to facilitate the discovering process. It is clear that the proposed covering and merging techniques support only subscription insertion, but not cancellation. As the covered and merged subscriptions are removed permanently during the discovery process, the owner of these subscriptions cannot receive any event if the coverer/merger subscription is removed.

The authors of [34] introduced a new data structure, called *RoXSum*, to aggregate the structural information of multiple XML documents in an efficient way that allows the subscription matching process to be applied directly on the aggregated content, instead of the original documents. The advantages of content aggregation and batch processing are combined to decrease communication costs and increase the performance of the message routing process. The idea of summarizing XML data is derived from the observation that elements within XML documents share structure and labels. RoXSum composes of two essential parts: a hierarchical data structure called *RoXSum tree*, and a set of document

identifiers called *RoXSum extents*. Each node in the RoXSum tree maps all structurally equivalent nodes from the document. For the purpose of message routing, a document identifier can be associated with only the RoXSum tree nodes that correspond to the leaf nodes of that document. Hence, after discovering the RoXSum tree nodes that satisfy a path query, it becomes straightforward to determine the documents that satisfy the query as well. The set of identifiers that correspond to a RoXSum tree node is the RoXSum extent of that node. Figure 7 presents an example RoXSum tree. The top of the figure shows a set of XML documents with identifiers *D1*, *D2*, *D3* and *D4*, while the bottom part illustrates the corresponding RoXSum tree. For example, the query */bib/book/last* on the documents shown in Figure 7 arrives. There is only one path in the RoXSum tree that matches this query. All documents within the extent of the RoXSum tree node *last* satisfy the query (i.e. documents with identifier *D1* and *D2*).

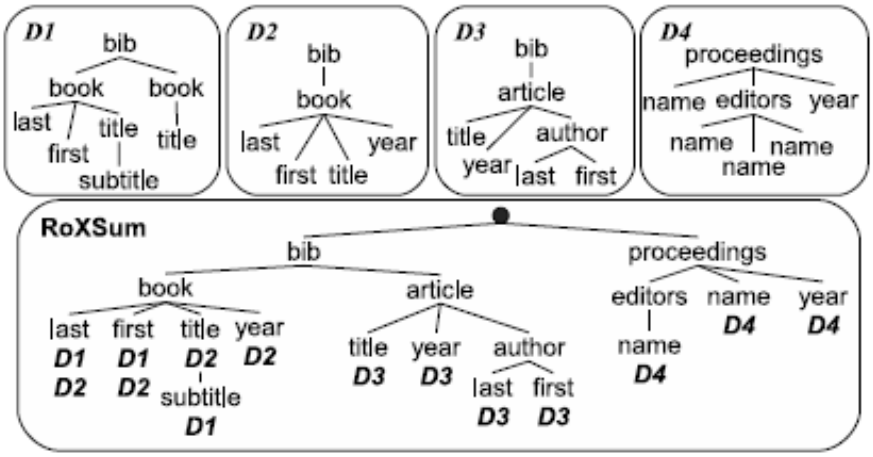


Figure 7: An example of RoXSum data structure [34]

New RoXSum trees are formed whenever a stream of XML documents is accepted by a router. RoxSum trees are built during the parsing process of the incoming XML stream. Each XML document is parsed in-order and either new index nodes or document identifiers are added to the RoxSum tree. Finally, the registered subscriptions are matched against the generated RoxSum trees, one RoxSum tree at a time.

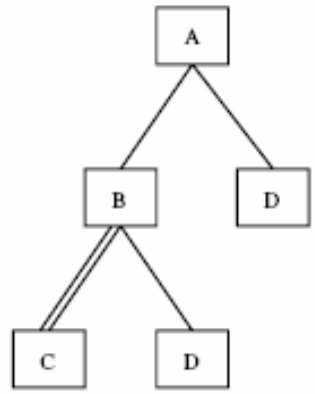
The proposed aggregation technique assumes that XML routers will receive the published documents in streams (batches). This assumption is not supported by many applications, therefore; the use of this technique is limited to a certain category of applications. The

total size of a RoXSum tree is proportional to the sum of the number of nodes in each document. Hence, large XML documents can dramatically affect the routing performance as they result in large RoXSum trees. The incoming streams of XML documents are processed independently. Accordingly, it is clear that the composition and decomposition algorithms for RoXSum are performed in an *incremental* process. However, this process has to be repeated at each router in the network for the same stream of XML documents. This may place a high burden on the XML routers and hence degrade their performance.

The authors of [31] proposed a new sequencing-based method, called *branch sequencing*, which converts an XML twig query into a branch sequence. In their work, the subscription queries are represented using a subset of XPath language called *twig patterns*. These are basically XPath expressions that include only parent/child, ancestor/descendent axes, and node predicates. The twig patterns are transformed into sequences and the matching process is performed using certain properties of the sequences. The proposed sequencing technique supports *holistic* matching of twig patterns with each input document as well as ordered twig patterns matching. Holistic means that a twig pattern is matched as a whole without breaking it into root-to-leaf paths. The subscription queries are parsed using an XPath parser and are converted into *branch sequences*, which are saved in a sequence store. After parsing the input XML documents, they go through a matching engine that matches them against the standing queries in the sequence store. The document nodes which satisfy a subscription query are saved in the buffer corresponding to the matched subscription. Afterward a separate predicate check is performed to identify those nodes that satisfy the predicates stored along with the query. Such nodes will be sent to the corresponding consumers. The proposed sequencing technique is slightly different from the one introduced by FiST [21] as it can retrieve the matched nodes in a single parse of the input document. In contrast, FiST has to parse the document twice, indexing the document nodes in the first parse.

To construct the branch sequence for a twig pattern, each node in the twig pattern has given a preorder number during the parsing process. When a leaf node is observed, the nodes from the closest branch node to this leaf node are output and this branch, excluding the branch point. A branch node is a node with two or more children. When the last leaf

node of the twig pattern is encountered, the nodes from the root of the pattern to the leaf node are output. This reflects the name of the proposed technique as the nodes are output branch by branch. As the twig patterns may include the ancestor-descendent axis, additional information is required about the relation between nodes. Information such as *sequence number*, *relation*, *position*, and *label* is stored along with the nodes to ensure that a particular branch node occurring in two or more positions in the sequence is matched to the *same* document node. The twig pattern will be eventually transformed to a sequence of *tuples* (also refereed to as *nodes*), which consist of a number of fields. Figure 8 shows a sample twig pattern and its corresponding sequence.



Twig Pattern: //A[B[//C]/D]/D

Seq. No.	1	1	1	1	1	1	1
Position	1	2	3	4	5	6	7
Label	B	C	A	B	D	A	D
Relation	#	D	#	C	C	#	C
Check–with	0	0	0	1	0	3	0

Branch Sequence of Twig Pattern

Figure 8: Sequencing a twig pattern [31]

An indexing mechanism for the twig pattern sequence nodes is used during the matching process. The sequence nodes are stored in a data structure, called *sequenceStore*, which can be considered as a two dimensional matrix where each row represents a sequence. For example, the *sequenceStore*[i, j] corresponds to the j*th* tuple in the branch sequence of the i*th* twig pattern. A hash-based index, called *sequenceIndex*, is used to index the twig pattern sequence nodes during the matching process. Here, the hash values of the sequence index are basically the different node labels. For example, the *sequenceIndex*[L] contains the node tuples, which have node label 'L'.

The proposed sequencing technique is not scalable for very large number of queries as it is expected to be memory intensive. The converting process for twig patterns results in

generating large number of tuples that need to be stored and evaluated with the incoming documents. This may increase the overhead on the routers and hence slow down the routing process. Dealing with redundant sequence tuples has not been addressed to reduce memory requirement and processing time. Incremental updating and cancellation of twig patterns is supported as they are transformed independently. The authors have not discussed the probability of the *false positive* that may occur due to the transformation process. However, it was stated in [21] that filtering using sequencing alone can lead to false positives.

ApproXFilter [35] proposed an approximative filtering algorithm, called *ApproXFilter*, for approximate filtering in a content-based routing system. Two complementary versions of the ApproXFilter algorithm are introduced for efficient filtering of large number of subscriptions: a time and a space optimized versions. Five steps are involved to match an ApproXFilter subscription query against an XML document. The first step is to transform all ApproXFilter subscription queries into their normalized form (i.e., Boolean disjunctions combined by conjunctions). The second step is to extend all subscription queries using the allowed predefined transformations such as deleting, inserting, and renaming parts of the queries using synonyms. The third step is to build a subscription match graph, called DAG (Directed Acyclic Graph), which represents all extended subscription queries. Every term (values and structures) in the extended query is interpreted as a graph vertex. Figure 9 shows an example Match DAG along with two extended query trees. The fourth step is to sequentially parse each incoming XML document and to concurrently traverse the Match DAG in depth-first order. Every difference to the original query is scored with additional costs. For each visited node in the Match DAG, the corresponding costs are calculated. The concept of costs can be seen as a similarity measurement between the documents and the matched subscriptions. A cost of zero corresponds to highest quality, which means exact matching. The higher the cost, the lower the quality. The fifth step is to notify the consumer about the matched document if the accumulated costs are less than a predefined threshold.

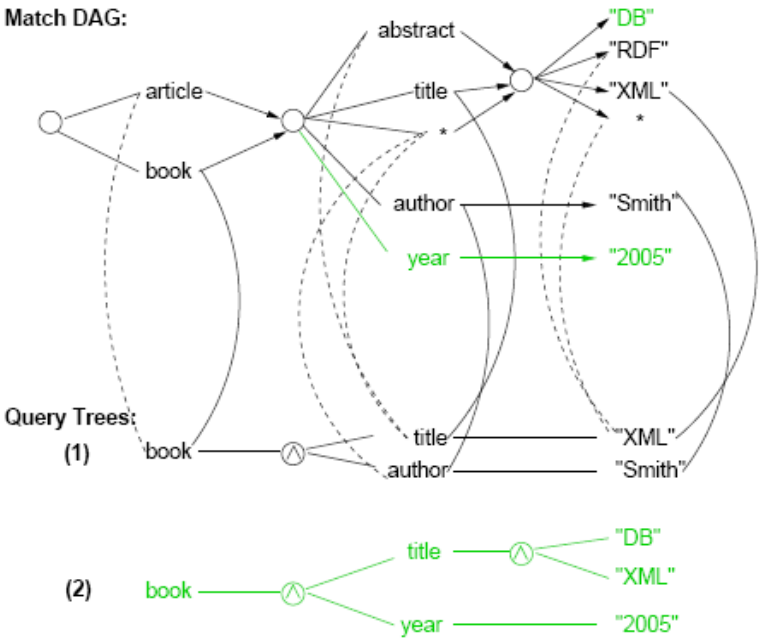


Figure 9: Concept of Match DAG and original query tree [35]

Two versions of algorithms are used to optimize time and space consumptions. Compact data structures along with a set of hashes are used to represent the extended query graph. Redundant node entries are not allowed in the proposed structures. As a result, each hash key is set only once into the graph and in the exact position for representing the original query structure. All costs are encoded only once. In the time-optimized version, the time required to evaluate a document is $O(n)$ while the space required is $O(n^2)$. In the space-optimized version, the required filter time is $O(n^2)$ while the space required is $O(n)$. Here, n reflects the number of vertices in the Match DAG.

The number of vertices in the DAG could grow considerably with a large population of subscriptions and terms (values, structures, and synonyms). Therefore, we expect that the Match DAG increases the overhead of the matching process as each input document has to be matched against a large number of vertices. We believe that the DAG in its current form does not support subscription cancellation since several subscriptions may share the same vertices in the match graph. In fact, after the updating process the DAG may no longer reflect the previously transformed subscriptions, thereby generating inappropriate matching results. Thus, rebuilding the DAG may become a necessary task to avoid this problem.

The work by [8] introduced a systematic study of subscription aggregation in which subscription specifications are expressed via a much more expressive model of *tree patterns* (subset of XPath expressions). In this study, a *tree-pattern* aggregation algorithm is proposed to aggregate an input set of tree patterns into a smaller set of generalized tree patterns in order to reduce their storage space requirements and to speed up the matching process. This algorithm depends on the use of document-distribution statistics to compute a precise set of aggregated tree patterns within a given space constraint and to minimize the probability of *false positives* (due to aggregation) during the filtering process. In order to aggregate an input set of tree patterns, the aggregation algorithm, presented in Figure 9, iteratively prunes the tree patterns by replacing a small subset of tree patterns with a more concise aggregate pattern, until a given space budget are met. During each iteration, a small set of candidate aggregate patterns are created and than the most promising candidate pattern is chosen (the one maximizing the gain in space while minimizing the loss in precision). The *least upper bound* (LUB) algorithm is proposed to compute the most precise aggregation tree pattern for a set of tree patterns. A containment algorithm also proposed to ensure the containment relationships between the original set of tree patterns and the aggregated tree pattern.

Algorithm AGGREGATE (S, k)
Input: S is a set of tree patterns, k is a space constraint.
Output: A set of tree patterns S' such that $S \sqsubseteq S'$ and $\sum_{p \in S'} |p| \leq k$.
1) Initialize $S' = S$;
2) **while** $(\sum_{p \in S'} |p| > k)$ **do**
3) $C_1 = \{x \mid x = \texttt{PRUNE}(p, |p| - 1),\ p \in S'\}$;
4) $C_2 = \{x \mid x = \texttt{PRUNE}(p \sqcup q, |p| + |q| - 1),\ p, q \in S'\}$;
5) $C = C_1 \cup C_2$;
6) Select $x \in C$ such that $Benefit(x)$ is maximum;
7) $S' = S' - \{p \mid p \sqsubseteq x, p \in S'\} \cup \{x\}$;
8) **return** S';

Figure 10: Tree pattern aggregation algorithm [8]

The problem domain addressed in the proposed technique focuses on transforming an input set of tree patterns into a smaller set. However, the aggregated technique does not reduce the number of the pattern entities in the routing table, which need to be evaluated against each input document. Furthermore, the issue of space constraint is addressed by

iteratively pruning the input set of tree patterns until the space requirement is met. It is not clear why the authors have not considered removing redundant tree patterns instead. To meet the objective of the proposed technique, three heavy, complementary processes need to be applied iteratively for each input set of tree patterns. These are aggregating (pruning) tree patterns, computing the most precise aggregate, and identifying the containment relationship of the output set. We thus suspect that with a large number of input patterns, the computation overhead incurred by these processes can substantially offset any performance gains of the proposed technique. Both subscription insertion and cancellation are performed in an incremental process; however, subscription cancellation in some cases may affect the benefit of the aggregation technique. If we consider the case when subscriptions do not stay long in the system after been registered, the overhead of the aggregation process would become very significant.

Concluding Remarks

This section summarizes the previous discussion on filtering and aggregating XPath-based subscriptions with an emphasis on lessons learned from investigating the proposed solutions. In particular, we highlight to which extent these techniques are effective and scalable when matching and updating a large number of XPath queries. For efficient data dissemination, more expressive subscription languages, such as XPath, have recently been used to express the consumers' interests. This has led to a marked increase in the complexity of content-based routing. As a result, much attention has been given to the performance challenges of routing XML data in the context of simple XPath-based queries.

Generally speaking, two optimization techniques are expected to provide a solid base for improving the performance and scalability of content-based routing: subscription *indexing* and *aggregation*. The former technique performs selective matching with only a compact subset of potentially matching subscriptions, while the latter transforms an original set of subscriptions into a compact and generalized subset of subscriptions to minimize the matching overhead. Table 1 present a simple comparison for the reviewed work with respect to: subscription insertions, subscription deletion, redundancy, matching accuracy, and scalability.

Project	Subscription Insertion	Subscription Deletion	Redundancy	Accuracy	Scalability
Bloom Filter [5]	Is performed in an incremental way	Is performed in an incremental way	Redundant transformation and matching of path queries	Negligible probability of false positives due to the hashing process	Is not an issue. It can filter millions of path queries.
Relational Graph [29]	Is performed in a *none* incremental manner due to the redundancy of covering relations	Is performed in a *none* incremental manner due to the redundancy of covering relations	Redundant covering relations among subscriptions that should be updated frequently	Probability of false negatives due to removing coverer subscriptions	Is scalable as it avoids unnecessary proliferation of subscriptions in the system
Piggyback [32]	Is performed in an incremental way	Is performed in an incremental way	Redundant annotation can be created for similar subscriptions stored in the routing table	Possibility of false positives due to the cancellation of subscriptions	Is scalable since the piggybacked data can reduce the matching overhead and network traffic
XRoute [4]	Is not incremental process as it needs to update covering relations and substitution links	Is not explained; however, we think it is not incremental as well for the same reason	Redundant subscriptions are identified during the building of the covering relations	Is claimed that the routing accuracy is free of false positives and negatives	Is limited as it is an expensive task to modify all existing relations in order to establish a new one
Subscription Tree [6]	Is not incremental process since the subscription tree needs to be partially rebuilt	Is not supported as it can affect the state of existing consumers	Redundant subscriptions are discarded during the covering and merging process	High probability of false negatives if coverer/merger subscriptions are cancelled	The sequential search for covering and merging relations limits the scalability
RoXSum Tree [34]	Is performed in an incremental way as the XML data is aggregated and not the subscriptions	Is performed in an incremental way for the same reason	Redundant creation of the *RoXSum* tree at each router and redundant queries in the routing table	Low probability of false positives due to the use of extent identifiers in the *RoXSum* tree	Is limited to small number of queries and documents due to iterative creation of the *RoXSum* tree
Sequencing [31]	Is performed in an incremental way as the twig patterns are transformed independently	Is performed in an incremental way for the same reason	Redundant sequence tuples are visible in the database	Probability of false positives due to filtering by subsequence matching alone	Is limited as it is a memory intensive due to generating a large number of tuples
ApproXFilter [35]	Is performed in an incremental way	Is not supported as many subscriptions may share similar vertices in the Match DAG	Redundant transformation of similar queries into the Match DAG	Low probability of false negatives may occur due to the normalization of subscriptions	Is the focus of the proposed Match DAG which shows its effectiveness in a limited test-bed
Tree Pattern [8]	Is performed in an incremental way	Is performed in an incremental way	Redundant nodes are defined and discarded from the generated patterns	Low probability of false positives due to aggregation	Is limited due to the heavy computation of the aggregation processes

Table 1: Comparison of aggregation techniques

From the reviewed work, we believe that Bloom filter is the most promising technique for improving the performance of content-based routing in the context of XML data and XPath-based subscriptions. This is due to its flexibility and scalability for filtering and matching a large number of path queries with a negligible probability of false positives. In addition, subscription insertion and deletion can be efficiently performed in an incremental manner. However, some aspects are missing that may further improve the performance of Bloom filter. The first aspect is to eliminate the existing redundancy among the routing table entries in the Bloom filter. This can result in minimizing the routing table and increasing the speed of the matching process. The second aspect is to avoid any redundant or unnecessary proliferation of subscriptions among the neighboring routers. This results in reducing the propagation overhead and the size of routing tables. We of course should keep in mind the performance and scalability of the chosen solutions when we approach these objectives.

References

1. World Wide Web Consortium W3C 2004a, *Extensible Markup Language (XML) 1.0.*, August 2004, Available from http://www.w3.org/XML/.
2. World Wide Web Consortium, *XML Path Language (XPath) Version 1.0.*, W3C recommendation November 1999, Available from http://www.w3.org/TR/xpath.
3. World Wide Web Consortium 2004b, *XML query language (XQuery) Version 1.0.*, W3C working draft October 2004, Available from http://www.w3.org/TR/xquery.
4. Chand, R. and Felber, P. 2004. XNET: A Reliable Content-Based Publish/Subscribe System. In *Proceedings of the 23rd IEEE international Symposium on Reliable Distributed Systems* (October 18 - 20, 2004). SRDS. IEEE Computer Society, Washington, DC, 264-273.
5. 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* (April 05 - 08, 2005). ICDE. IEEE Computer Society, Washington, DC, 890-901.
6. Guoli Li Shuang Hou Jacobsen, H.-A., XML Routing in Data Dissemination Networks, In *Proceedings of the 23rd IEEE International Conference on Data Engineering*, (April 2007), 1400-1404.
7. Chand, R. and Felber, P. A. 2003. A Scalable Protocol for Content-Based Routing in Overlay Networks. In *Proceedings of the Second IEEE international Symposium on Network Computing and Applications* (April 16 - 18, 2003). NCA. IEEE Computer Society, Washington, DC, 123.

8. Chee-Yong Chan, Wenfei Fan , Pascal Felber, Minos Garofalakis, Rajeev Rastogi. 2002. Tree Pattern Aggregation for Scalable XML Data Dissemination. In *Proceedings of the 28th VLDB Conference*, (Hong Kong, China 2002), 826-837.

9. SensorML homepage, http://vast.uah.edu/SensorML/home.html

10. Transducer Markup Language homepage, http://www.transducerml.org/index.htm

11. Altinel, M., Franklin, M. J., Efficient Filtering of XML Documents for Selective Dissemination of Information. Proceedings of the 26th VLDB Conference, Cairo, Egypt, 2000.

12. Diao et al., Path Sharing and Predicate Evaluation for High-Performance XML Filtering, ACM Transactions on Database Systems, Vol. 28, No. 4, December 2003, Pages 467–516.

13. Green, T., Gupta, A., Miklau, G., Onizuka, M., and Suciu, D. 2004. Processing XML streams with deterministic automata and stream indexes. *ACM Transactions Database Systems.* 29, 4 (Dec. 2004), 752-788.

14. Candan, K. S., Hsiung, W-P., Chen, S., Tatemura, J., Agrawal, D. AFilter: Adaptable XML filtering with Prefix-Caching and Suffix-Clustering, Proceedings of the 32nd VLDB Conference, 2006.

15. Ives, Z., Halevy, A. Y., and Weld, D. S. 2002. An XML query engine for network-bound data. *The VLDB Journal* 11, 4 (Dec. 2002), 380-402.

16. Florescu, D., Hillery, C., Kossmann, D., Lucas, P., Riccardi, F., Westmann, T., Carey, M. J., Sundararajan, A., and Agrawal, G. 2003. The BEA/XQRL streaming XQuery processor. In *Proceedings of the 29th international Conference on Very Large Data Bases - Volume 29* (Berlin, Germany, September 09 - 12, 2003). J. C. Freytag, P. C. Lockemann, S. Abiteboul, M. J. Carey, P. G. Selinger, and A. Heuer, Eds. Very Large Data Bases(Vldb) Series. VLDB Endowment, 997-1008.

17. Bar-Yossef, Z., Fontoura, M., and Josifovski, V. 2005. Buffering in query evaluation over XML streams. In *Proceedings of the Twenty-Fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Baltimore, Maryland, June 13 - 15, 2005). PODS '05. ACM, New York, NY, 216-227.

18. Bar-Yossef, Z., Fontoura, M., and Josifovski, V. 2007. On the memory requirements of XPath evaluation over XML streams. *J. Comput. Syst. Sci.* 73, 3 (May. 2007), 391-441.

19. Chan, C., Felber, P., Garofalakis, M., and Rastogi, R. 2002. Efficient filtering of XML documents with XPath expressions. *The VLDB Journal* 11, 4 (Dec. 2002), 354-379.

20. Bruno, N., Koudas, N., and Srivastava, D. 2002. Holistic twig joins: optimal XML pattern matching. In *Proceedings of the 2002 ACM SIGMOD international Conference on Management of Data* (Madison, Wisconsin, June 03 - 06, 2002). SIGMOD '02. ACM, New York, NY, 310-321.

21. Kwon, J., Rao, P., Moon, B., and Lee, S. 2005. FiST: scalable XML document filtering by sequencing twig patterns. In *Proceedings of the 31st international Conference on Very Large Data Bases* (Trondheim, Norway, August 30 - September 02, 2005). Very Large Data Bases. VLDB Endowment, 217-228.

22. Nicolas Bruno, Luis Gravano, Nick Koudas, Divesh Srivastava 2003. Navigation- vs. Index-Based XML Multi-Query Processing. In *proceedings of the 19th International Conference on Data Engineering (ICDE'03)*, 139.

23. Chen, Y., Davidson, S. B., and Zheng, Y. 2006. An Efficient XPath Query Processor for XML Streams. In *Proceedings of the 22nd international Conference on Data Engineering* (April 03 - 07, 2006). ICDE. IEEE Computer Society, Washington, DC, 79.

24. S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In Proceedings of the 2002 International Conference on Data Engineering, 2002.

25. C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data, 2001.

26. Koch, C., Scherzinger, S., Schweikardt, N., and Stegmaier, B. 2004. Schema-based scheduling of event processors and buffer minimization for queries on structured data streams. In *Proceedings of the Thirtieth international Conference on Very Large Data Bases - Volume 30* (Toronto, Canada, August 31 - September 03, 2004). M. A. Nascimento, M. T. Özsu, D. Kossmann, R. J. Miller, J. A. Blakeley, and K. B. Schiefer, Eds. VLDB Endowment, 228-239.

27. Peng, F. and Chawathe, S. S. 2003. XPath queries on streaming data. In *Proceedings of the 2003 ACM SIGMOD international Conference on Management of Data* (San Diego, California, June 09 - 12, 2003). SIGMOD '03. ACM, New York, NY, 431-442.

28. Bloom, B. H. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (Jul. 1970), 422-426.

29. Zhenhui Shen, Srikanta Tirthapura, and Srinivas Aluru. 2005. Indexing for subscription covering in publish-subscribe systems. In *Proceedings of IEEE International Conference on Data Engineering*. (ICDE'05), 32-43.

30. Aneesh Raj and P Sreenivasa Kumar. 2007. Branch Sequencing Based XML Message Broker Architecture. *N Proceedings of the IEEE 23rd International Conference on Data Engineering* (Istanbul, Turkey, April 16-20, 2007). ICDE 2007, 656-665.

31. Hou, S. and Jacobsen, H. 2006. Predicate-based Filtering of XPath Expressions. In *Proceedings of the 22nd international Conference on Data Engineering* (April 03 - 07, 2006). ICDE. IEEE Computer Society, Washington, DC, 53.

32. Chan, C. Y. and Ni, Y. 2007. Efficient xml data dissemination with piggybacking. In *Proceedings of the 2007 ACM SIGMOD international Conference on Management of Data* (Beijing, China, June 11 - 14, 2007). SIGMOD '07. ACM, New York, NY, 737-748.

33. Raphael Chand, Pascal Felber, "Scalable Distribution of XML Content with XNet," *IEEE Transactions on Parallel and Distributed Systems*, 19, 4 (April 2008), 447-461.

34. Zografoula Vagena, Mirella Moura Moro, Vassilis J. Tsotras. 2007. RoXSum: Leveraging Data Aggregation and Batch Processing for XML Routing. ICDE (April 2007), 1466-1470.

35. Hinze, A., Michel, Y., and Schlieder, T. 2006. Approximative filtering of XML documents in a publish/subscribe system. In *Proceedings of the 29th Australasian Computer Science Conference - Volume 48* (Hobart, Australia, January 16 - 19, 2006). V. Estivill-Castro and G. Dobbie, Eds. ACM International Conference Proceeding Series, vol. 171. Australian Computer Society, Darlinghurst, Australia, 177-185.

36. Boone P. 2007. A Hybrid XML Filtering Engine for Publish/Subscribe Content-Based Routing. Carleton University, Ottawa, January 2007.