Scalable Filtering of XML Data for Web Services

P. Felber
Institut EURECOM
06904 Sophia Antipolis, France

C.-Y. Chan, M. Garofalakis, R. Rastogi
Bell Labs, Lucent Technologies
Murray Hill, NJ 07974, USA

Abstract

With the advent of XML as the de facto language for data interchange and the rapid emergence of Web services, Web applications have to deal with increasing amounts of XML traffic, such as SOAP requests. In order to scale to large number of clients and high data traffic, back-end systems to which Web services are mapped face the challenging task of efficiently filtering, classifying, and routing incoming XML requests to the appropriate component in the infrastructure. As the destination of each XML request depends on its actual type and content, Web service applications have to incorporate highly-efficient content-based routing *technology. In this paper, we address the issue of scalable filtering of XML data. We describe sophisticated techniques for matching data against large number of tree-structured filters expressed using the XPath language. We propose a hierarchical XML routing architecture that supports extremely high loads, and we present experimental results that demonstrate its performance.*

Keywords: XML, XPath, content-based routing, filtering.

1 Introduction

The Extensible Markup Language (XML) [7] is emerging as the universal format for exchanging structured data over the Internet. In particular, as the Web becomes more and more used for application to application communication, an increasing number of Web service applications are being deployed and make application functionality available over the Internet in a standardized, programmatic manner. Web services utilize XML-based open standards, such as WSDL [9] for service definition and SOAP [8] for service invocation. The services can be reached using the SOAP protocol, and requests can be properly constructed using the information available in the WSDL description present with the Web service. A wide range of domain-specific specifications are also based on XML, e.g., for business to business interactions or financial data. Even HTML, arguably the most widely used data format in the Internet, has been recently rewritten as an XML-based specification [6].

The rapid growth of XML traffic associated with Web services motivates the need for scalable mechanisms to filter, classify, and route XML message. Enterprise application servers, for instance, must scale to large numbers of clients, high throughput, and support a variety of XML-based protocols (see Figure 1). XML data is received by a Web server (generally associated with a firewall) and filtered by one or several XML routers. These routers are responsible for dispatching XML data, according to its type or content, to the appropriate backend server, possibly using some elaborate scheme such as load-balancing or selective multicast. XML routers can also act as a sophisticated firewall, by filtering out unauthorized or invalid XML messages[3, 4].

With content-based routing, the recipient of an XML message depends on the actual type and content of the data. Therefore, back-end servers can be specialized to process only some type of XML data in a very efficient manner. For instance, some server

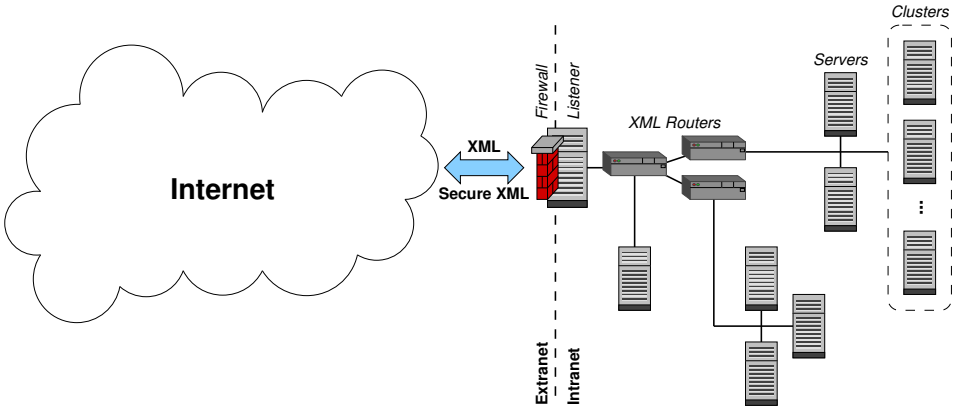


Figure 1: XML Routing for Enterprise Web Servers.

could be responsible for all SOAP requests, or only for SOAP requests related to stock quotes, or just for the stock quotes of some companies. The resulting services become more scalable and cost-effective than those based on traditional load-balancing schemes, which require each server to be able to process any type of request. Although not specific to Web services, efficient techniques for content-based XML data routing are of great importance in that context because of the large amount of XML traffic associated with Web services.

2 XML for Web Services

Web services provide an extensible and interoperable framework for application to application communication, thanks to the use of universal data formats and protocols, such as XML and XPath.

2.1 XML

The Extensible Markup Language (XML) is emerging as the universal format for exchanging (semi-)structured data over the Internet and increasing amounts of data are made available in XML format. Because of its simple structure, XML is easy to interpret and process by applications. By being vendor- and platform-neutral, as well as agnostic about how content appears, XML makes it simple to integrate existing applications and to represent data in various human-readable formats.

XML defines an unambiguous mechanism for constraining structure in a stream of information. XML documents can optionally include a type definition (XML schema or DTD[1]), which defines the document structure by describing its legal building blocks.

Consider the following sample XML document (example 5 of [8]) that represents a SOAP request asking the current value of the stock of a company.

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://.../soap/envelope/"
  SOAP-ENV:encodingStyle="http://.../soap/encoding/">
    <SOAP-ENV:Header>
        <t:Transaction
          xmlns:t="some-URI"
          SOAP-ENV:mustUnderstand="1">
            5
        </t:Transaction>
    </SOAP-ENV:Header>
    <SOAP-ENV:Body>
        <m:GetLastTradePrice xmlns:m="Some-URI">
            <Symbol>DEF</Symbol>
        </m:GetLastTradePrice>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The document structure is specified by the means of start and end tags (tags are enclosed between <

[1]Note DTDs are becoming deprecated in favor of XML schemas, and are explicitly disallowed for defining Web service message formats.

and >, and end tags start with /). Start tags contain an optional list of attributes, which are essentially key-value pairs. Text data can be enclosed between tag pairs, as between the Symbol tags. We will refer to this text as the tag's value or content.

XML documents have a hierarchical structure as illustrated by the tree-based representation of our sample XML document in Figure 2(a). Attributes (with their names prefixed by the symbol @) are represented as children of their associated tag; values (shown within quotation marks) are represented as children of their associated attribute or tag.

Roughly speaking, we can distinguish between structural elements—tags and attributes—and the actual data values associated with these elements. Connections between structural elements are represented using solid lines, and between elements and values by dashed lines. Intuitively, the *type* of an XML document is defined by its tags and attributes, while the data associated with its tags and attributes define its *value*.[2] By disconnecting the dashed lines from the tree representation of an XML document, we obtain its type. Two SOAP request asking the value of different stocks would have the same type, but different values. Figures 2(b) to (d) show additional XML tree examples.

2.2 XPath

The structured, extensible nature of XML allows for a powerful combination of *type-based* and *value-based* content filtering. To that end, XML filters should be able to express constraints on both the type and the value of data. Because of its simplicity and standardization, the W3C XPath [5] addressing language—or a subset thereof—is the most widely used for that purpose.

XPath treats XML documents as a tree of nodes and offers an expressive way to specify and select parts of this tree. An XPath expression contains one or more *location steps*, separated by slashes (/). In its more basic form, a location steps designate an element name followed by zero or more predicates specified between brackets. Predicates are generally specified as constraints on the presence of structural elements, or on the values of XML documents using basic comparison operators ($=$, $<$, $\leq$, $>$, $\geq$). XPath also allows the use of wildcard (*) and ancestor/descendant (//) operators, which respectively match exactly one and an arbitrarily long sequence of element names. In the context of data filtering, an XML document *matches* an XPath expression when the evaluation of the expression yields a non-null object.

Consider the following sample XPath expressions.

```
(i):   //m:GetLastTradePrice/Symbol
(ii):  //m:GetLastTradePrice/Symbol[text()="DEF"]
(iii): /SOAP-ENV:Envelope/SOAP-ENV:Body/*//Symbol[text()="DEF"]
(iv):  //t:Transaction/@SOAP-ENV:mustUnderstand
(v):   //t:Transaction[@SOAP-ENV:mustUnderstand=1]
(vi):  //Stock/Symbol[text()="GHI"]
(vii): //Stock/Price[text()>15]
(viii)://Stock[Symbol/text()="GHI"][Price/text()>15]
```

XPath expression (i) designates documents that have two consecutive nodes m:GetLastTradePrice and Symbol at any level in the document (the initial // specifies that any number of nodes can appear before the first element). XPath expression (ii) additionally constraints the value of the Symbol node to be equal to "DEF" (text() selects the text node below the current node). Both of these expressions match the XML document in Figure 2(a). XPath expression (iii) designates SOAP messages that have a Symbol node with value "DEF" at least two levels deep inside the message's body (below the Body node). Note that this expression specifies an absolute path from the root of the XML document. It matches the XML documents in Figures 2(a), (b), and (d).

Attributes are specified in XPath expressions in a very similar way to tag nodes, but are prefixed with @. XPath expression (iv) designates documents that have a Transaction node with a SOAP-ENV:mustUnderstand attribute, and XPath expression (v) further mandates this attribute to have the value "1".

XPath can also be used to express more complex filters where the structural constraints are not limited to single paths. Consider a filter that selects SOAP messages with quotes for symbol "GHI" that have a price higher than "15". The first constraint

[2]Formally, the type of an XML document is specified by an associated schema.

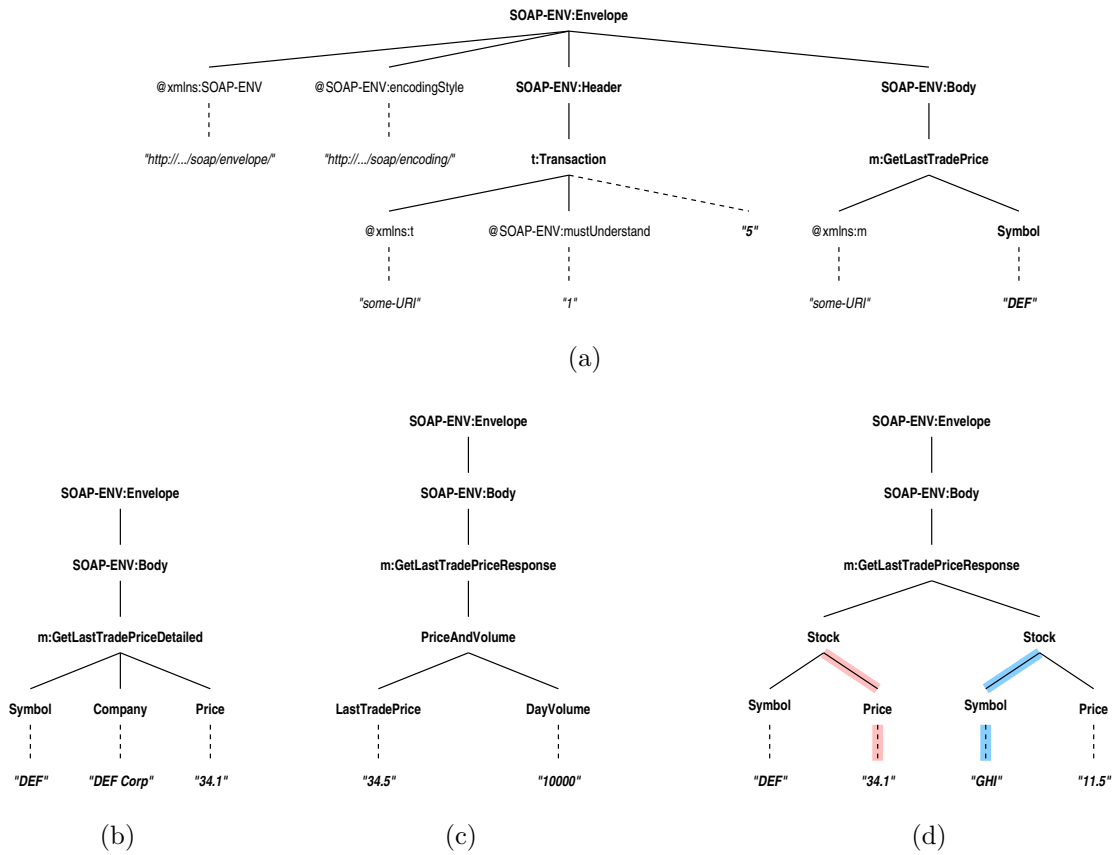


Figure 2: XML trees for sample SOAP messages. (a), (b) and (c) Examples 5, 6 and 8 of [8] (attributes have been omitted). (d) SOAP message with multiple response values.

can be expressed with XPath expression (vi), and the second one with expression (vii). A simple conjunction of these two XPath expression is not sufficient, however, to obtain the desired filter: the document in Figure 2(d) contains both expressions (matching paths are highlighted), but the price that matches the second expression is not that of symbol `"GHI"`. The correct filter must further constrain the matching `Symbol` and `Price` nodes to share the same `Stock` parent node. Such structural constraints are achieved using *tree-structured* expressions, which are expressed in XPath by defining multiple predicates on the same node. XPath expression (viii) is one possible embodiment of the desired filter.

3 Scalable Architectures for Web Services

The combination of semi-structured data and tree-structured filters offers a very flexible and expressive framework for content-based routing, when compared to the traditional keyword-based information retrieval techniques used for unstructured data. It does however also increase the complexity of filtering and makes efficient matching algorithms a prerequisite to scalable content routing.

3.1 Efficient Filtering of XML Data

We have developed a novel index structure, termed XTrie, that supports the efficient filtering of XML documents based on XPath expressions. Our XTrie index structure offers several novel features that make it especially attractive for Web Service applications with strong scalability and performance requirements. First, XTrie is designed to support effective filtering based on complex, tree-structured XPath expressions (as opposed to simple, single-path specifications). Second, the XTrie structure and algorithms are designed to support both ordered and unordered matching of XML data. Third, by indexing on sequences of elements organized in a trie structure and using a sophisticated matching algorithm, XTrie is able to both reduce the number of unnecessary index probes as well as avoid redundant matchings, thereby providing extremely efficient filtering.

The XTrie for a given set of XPath expressions is constructed as follows. Each XPath expression is first decomposed into a minimal number of substrings, where a substring is a non-empty sequence of elements that are separated by the parent/child operator (`/`) that is optionally prefixed by an ancestor/descendant operator (`//`) and wildcards (`*`). The collection of decomposed substrings are then organized using a sophisticated trie structure and an auxiliary table. The trie allows enables both space-efficient indexing as well as time-efficient retrieval of XPath expressions, while the table stores additional information used for detecting valid matches.

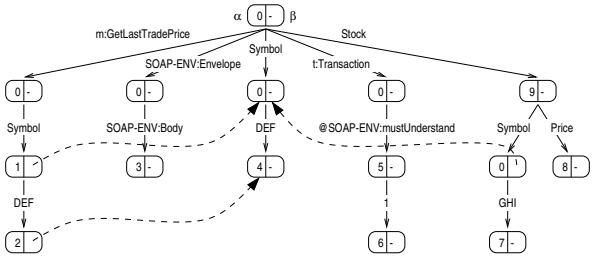
The trie is a rooted tree constructed from the set of distinct substrings in the given set of XPath expressions, where each edge is labeled with some element name. As the trie factorizes substrings with common prefixes, its size generally remains small. Each node in the trie has pointers to other nodes and to rows in the auxiliary table. The table contains one row for each substring of each indexed XPath expression. Each row has a set of values that describes the positional and structural constraints of the associated substring. Figure 3 shows an XTrie index for the sample XPath expressions in Section 2.2.

Informally, the matching algorithm of XTrie on an incoming XML document works as follows: the XML document is first parsed using the event-based SAX parser, which reports occurrences of XML elements (start tags, text, etc.) as the XML document is being parsed. The matching algorithm tries to map sequences of start tags, attributes, and text values to paths in the trie by following its edges. For each matching substring detected, the auxiliary table is used to verify the substring's positional constraints with respect to its previously-matched parent and sibling substrings, as well as any associated predicates. Information about partially-matched XPath expressions is maintained at runtime in a data structure that stores the occurrence, depth, and scope of substrings previously encountered. When end tags are parsed, the runtime information is updated to invalidate substring matches that are "out of scope". An XPath expression is completely matched when all its substrings have been encountered with their associated constraints and predicates validated. Note that by using the event-based SAX parser, which does not require the construction of an in-memory representation of the input document for matching, XTrie is capable of filtering streaming XML data as well. An exhaustive description of the XTrie algorithms can be found in [2].

3.2 Parallel XTrie

Large Web service applications usually impose very demanding performance requirements as they need to handle large numbers of XPath expressions and process huge amounts of requests. An effective approach to improve the scalability of XTrie is to parallelize its processing so that the time- and space-consuming task of filtering data can be shared by multiple XML routers. Parallelization can be easily achieved by using a cluster of XML routers organized according to one of two simple strategies.

- *Sharing XML workload ("data-sharing" strategy).* Each XML router in the cluster manages the complete set of XPath expressions, and each XML document is dispatched to only one router according to some load-balancing strategy (Figure 4(a)).
- *Sharing XPath expressions ("filter-sharing"*



(a)

	Parent Row	Rel Level	Rank	Num Child	Next	
1	0	[2, ∞]	1	0	0	//m:GetLastTradePrice/Symbol
2	0	[3, ∞]	1	0	0	//m:GetLastTradePrice/Symbol/DEF
3	0	[2, 2]	1	1	0	SOAP-ENV:Envelope/SOAP-ENV:Body
4	3	[3, ∞]	1	0	0	*//Symbol/DEF
5	0	[2, ∞]	1	0	0	//t:Transaction/@SOAP-ENV:mustUnderstand
6	0	[3, ∞]	1	0	0	//t:Transaction/@SOAP-ENV:mustUnderstand/1
7	0	[3, ∞]	1	0	10	//Stock/Symbol/GHI
8	0	[2, ∞]	1	0	11	//Stock/Price
9	0	[1, ∞]	1	2	0	//Stock
10	9	[3, 3]	1	0	0	//Stock/Symbol/GHI
11	9	[2, 2]	2	0	0	//Stock/Price

(b)

Figure 3: XTrie for the sample XPath expressions of Section 2.2. (a) Trie. (b) Auxiliary table.

strategy). XPath expressions are shared equally among all XML routers in such a way that each distinct expression is managed by exactly one router (Figure 4(b)). Incoming XML documents are filtered by all the routers.

Note that the above two strategies are designed for optimizing different scalability requirements. The first strategy maximizes the filtering throughput by enabling the maximum number of documents to be concurrently processed. In contrast, by processing each input document with all the routers in the cluster (with each router responsible for a small and disjoint subset of XPath expressions), the second strategy minimizes the filtering latency time.

3.3 Hierarchical XTrie

One approach to obtain both reasonable filtering throughput as well as filtering response it to try to combine the strengths of the above two strategies into a hybrid strategy that organizes the cluster of XML routers into a hierarchical configuration. Each incoming XML document is first sent to the root router, which performs a very coarse filtering to decide the subset of its child routers to send the document to for more refined filtering. This top-down propagation and filtering of the XML document continues from one level to the next until the document reaches the leaf level, where it is filtered by a subset of leaf routers to decide the target backend servers to dispatch the document to.

An example of this hierarchically organized XTrie is shown in Figure 4(c), where each router manages a set of filters (i.e., XPath expressions). Leaf routers are organized using the filter-sharing strategy, so that each leaf router manages a disjoint subset of the workload of XPath expressions (referred to as *end filters*). Each internal router manages a collection of sets of *intermediate filters*, with one set corresponding to each of its child routers; each set of intermediate filters provides a coarse-level "summary" of the set of filters managed by its corresponding child router. Specifically, each filter F in a child router must be "covered" by some filter in F' in its parent router such that any XML document that matches F will also match F'. For instance, the XPath expression `//Symbol` covers each of the XPath expressions (i), (ii), (iii), (vi), and (vi) in Section 2.2. This property guarantees that whenever a document matches some end filter in some leaf router, the document will always be routed to that leaf router by the internal routers. This "hierarchical" strategy therefore combines the advantages of the filter-sharing and data-sharing strategies: like the former, each document is being processed by multiple routers to improve filtering latency; and like the latter, multiple documents can be processed concurrently to improve filtering throughput.

The main challenge of the hierarchical strategy lies in the clustering of XPath expressions into subsets to be assigned to leaf XML routers, and the clustering of the collection of XML routers at one level into

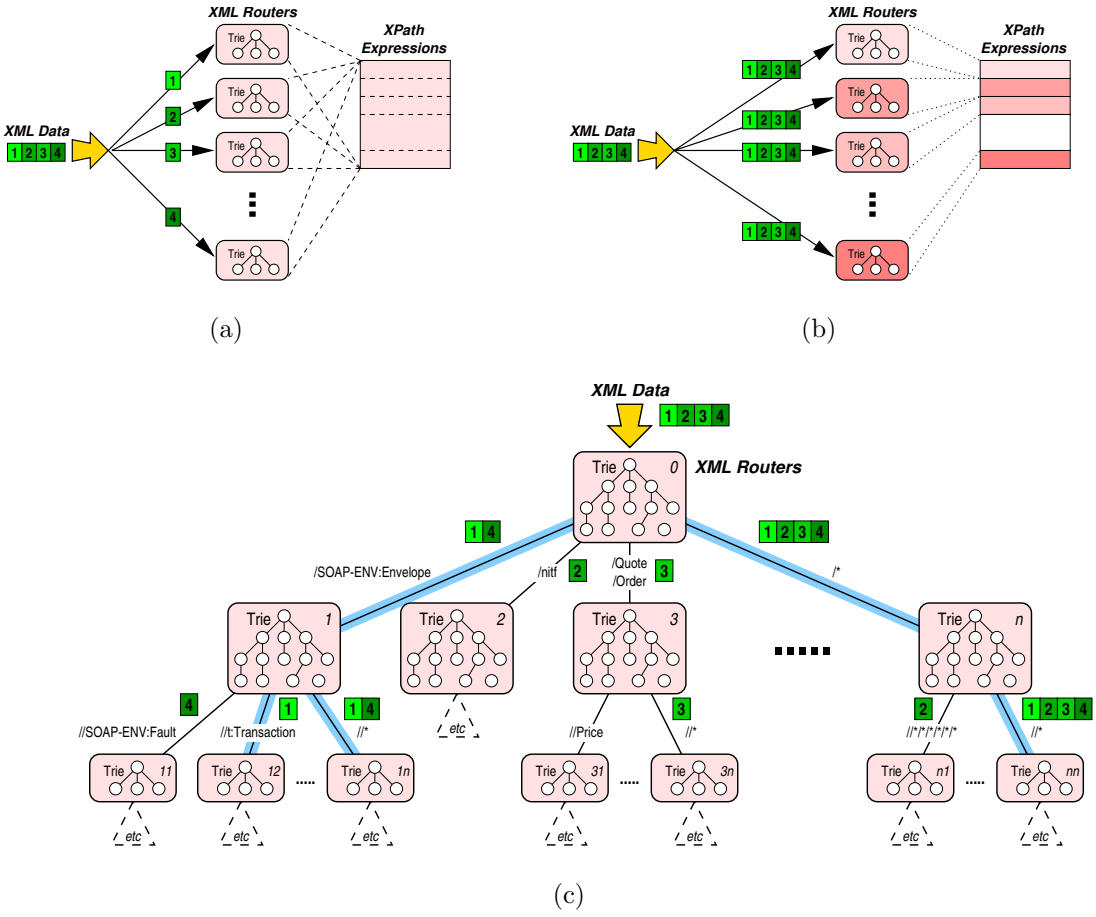


Figure 4: Strategies for parallel filtering of XML documents with XTrie. (a) Sharing XML workload. (b) Sharing XPath expressions. (c) Hierarchical filtering (highlighted paths are traversed by the sample SOAP message of Section 2.1).

subsets to be assigned to parent routers at the level above. This clustering must be performed in such a way that (1) each subset can be represented concisely by a small set of intermediate filters installed in the parent router, and (2) each XML document is propagated down to only a small subset of leaf routers. The first condition ensures efficient filtering at internal routers since each of them manages only a small set of filters, while the second condition optimizes the filtering throughput by ensuring that only the relevant subset of routers are used to process each XML document.

Clustering XPath Expressions. There are a number of simple ways that can be used to cluster XPath expressions. One straightforward approach is to partition absolute XPath expressions based on the element names of their root nodes. This clustering technique is effective because XML documents that have distinct root nodes are guaranteed to have different types (schema). An example of this partitioning is illustrated in the first level of the hierarchical XTrie in Figure 4(c). The topmost router filters XML documents according to their root tags and forward them to the appropriate subset of child routers

downstream. For instance, router 1 is responsible for all SOAP requests, and router 3 is responsible for both quote and order requests. The sub-tree at router n contains all XPath expressions that do not have an explicit root tag (i.e., they begin with `//` or `/*`). Thus, in Figure 4(c), each XML document will always be propagated to router n, as well as to any router associated with the document's root tag. More generally, non-absolute XPath expressions can be partitioned using element names that are specific to individual schemas.

Clustering XPath expressions according the type of the XML documents they refer to is very effective in practice, but it only permits building a coarse grained router hierarchy. This may prove inefficient when, for instance, some types of document occur much more frequently than others. It is therefore desirable to also cluster XPath expressions in the context of a single schema. For instance, router 11 in Figure 4(c) manages SOAP error messages, and router 12 filters SOAP messages that have transaction identifiers. An approach that yields good results is to cluster XPath expressions according to "exclusive" sets of element names that never or rarely appear in the same XML document. Finding exclusive elements is straightforward when the XML schemas are known in advance; it can also be achieved by observing XML data flowing through the routers and gathering information about the patterns that are unlikely to occur in the same document.

Generating Intermediate Filters. Unlike the end filters managed by leaf routers, which correspond to the input set of XPath expressions, the intermediate filters at each internal router are "artificially" constructed to *cover* the set of XPath expressions in its child routers. In order to provide effective coarse-level filtering, the set of intermediate filters need to satisfy two conflicting requirements: (1) it should be small to enable fast filtering; and (2) it should provide a *tight covering* in the sense that it should minimize the number of documents matching some intermediate filter in a router but not matching any filters in its corresponding child router. In other words, it should minimize the unnecessary forwarding of documents to irrelevant downstream XML routers.

When XPath expressions are clustered according to the element names, the construction of the intermediate filters is trivial. However, when the set of XPath expressions managed by a router is more diverse, finding a set of intermediate filters is a challenging task. In [1], we have proposed an efficient aggregation algorithm that makes effective use of document-distribution statistics in order to compute a *precise* and *compact* set of coarse-level XPath expressions for a given set of XPath expressions. This algorithm can be straightforwardly applied to the generation of intermediate filters. When the set of XPath expressions to aggregate have strong similarities—which is expected from an effective clustering scheme—our aggregation algorithm can produce intermediate filters several orders of magnitude smaller without significant loss in precision.

4 Performance Evaluation

To test the effectiveness of our XML routing technology for Web services, we conducted an extensive performance study of the XTrie filtering algorithm using real-life document types and large numbers of tree patterns. Based on these results, we evaluated the performance of the various architectures for parallelizing XTrie. Because the filtering time of an XML message is several orders of magnitude slower than its actual transmission over the network, we did not take into account the transmission time. Our experimental study thus shows the asymptotic performance of the parallel XTrie architectures. Experiments were conducted on a 1.5 GHz Intel Pentium 4 machine with 512 MB of main memory running Linux.

For the experiments presented here, we selected 10 real-world document types used in major commercial applications, with the number of distinct elements ranging from 77 to 2727 elements, and with up to 8512 distinct attributes. Most of these types have recursive structures, which can be nested to produce XML documents with arbitrary number of levels. For each type, we generated a representative set of XML documents with approximately 100 tags and 20 levels, as well as sets of XPath expressions containing

approximately 10% of * and // operators.

We ran experiments with two variants of the XTrie algorithm: the first variant is optimized for single-path XPath expressions, where each node has at most one child; the second variant is optimized for ordered matching of tree-structured XPath expressions. We compared the scalability of XTrie parallelization for the three strategies presented in Section 3: the data-sharing, the filter-sharing, and the hierarchical strategies. The results shown were obtained from simulation, based on the experimental results of the XTrie algorithms.

4.1 Raw XTrie Performance

The performance of the non-parallel XTrie algorithms decreases *linearly* with the number of XPath expressions, as well as with the length of the XML documents. In particular, the XTrie variant optimized for single-path XPath expressions filters around 100 messages per second with $20,000$ XPath expressions, and 27 messages with $200,000$ XPath expressions. The performance of the variant for tree-structured expressions is approximately one order of magnitude slower. Detailed experimental results can be found in [2].

4.2 Data-Sharing and Filter-Sharing Strategies

Figure 5(a) compares the throughput performance of the data-sharing and filter-sharing strategies (with logarithmic scales) for $200,000$ XPath expressions. As expected, the throughput of the data-sharing strategy scales linearly with the number of XML routers. In contrast, the throughput of the filter-sharing strategy increases more gradually with the number of XML routers until a maximum throughput of 170 messages per second (since all XML routers filter each message, throughput is limited by constant cost of parsing XML documents, irrespective of the numbers of XPath expressions managed by each router).

A comparison of the latency performance of the parallel XTrie strategies is shown in Figure 5(b) for the same set of $200,000$ XPath expressions. For the data-sharing strategy, increasing the number of XML routers does not improve the latency performance since only one of the routers is used to filter each incoming document. On the other hand, for the filter-sharing strategy, the filtering latency decreases proportionally to the number of XPath expressions managed per router.

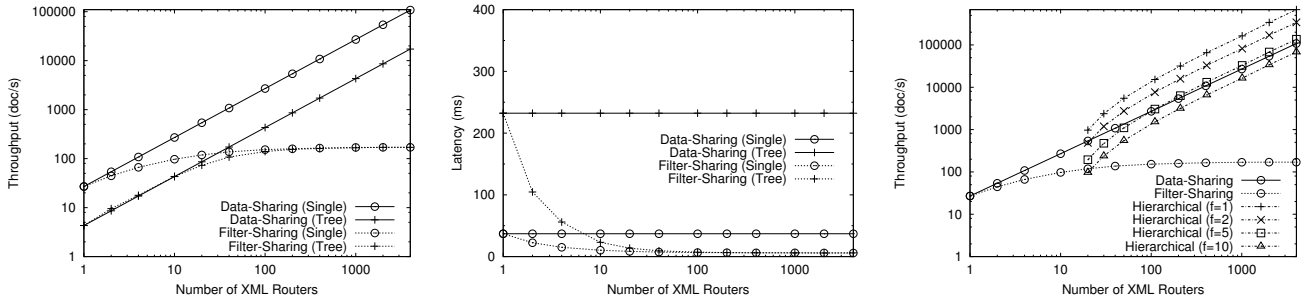
Our performance results clearly validate our expectations about the strengths of the different parallelization strategies. The appropriate strategy to adopt for a Web service application depends on the desired performance objective.

4.3 Hierarchical Strategy

We evaluated the performance of the hierarchical strategy with absolute XPath expressions (i.e., they are relative to the root node) and a clustering scheme where the first-level router filters documents based on their root tags. This initial routing step is extremely efficient because XML documents do not need to be parsed (the root tag is the first element that appears in a document) and the filtering degenerates into a simple hash-table lookup. As all the input XPath expressions are absolute, an XML document is routed to at most one subtree during the first-level routing.

In practice, two major factors prevent hierarchical filtering from being optimal. First, the average number of leaf routers ultimately reached by each XML document—its *fanout*—is generally greater than 1 and reduces parallelism. Second, there may be contention on some leaf routers that are traversed more often than others. To account for these factors, we evaluated the performance of the hierarchical configuration with different values for the average fanout f. Results are shown in Figure 5(c). Although the hierarchical strategy requires additional routers for intermediate filtering (we used at least 20 in our experiments), it quickly achieves better throughput than the other strategies. The ideal case with $f = 1$ scales the best; the data-sharing strategy should be preferred for fanouts higher than 5.

A promising alternative, in practice, is to combine the hierarchical and data-sharing strategies. A simple approach consists of using a one-level routing hierarchy that partitions the XPath expressions based



(a) (b) (c)

Figure 5: Parallel XTrie filtering performance with 200,000 single-path XPath expressions. (a) Throughput. (b) Latency. (c) Throughput of hierarchical filtering (single-path XTrie algorithm).

on their root nodes, and then load-balancing XML documents, according to the data-sharing strategy, to clusters of XML routers dimensioned according to the document distribution. As previously mentioned, the first-level filtering phase is extremely efficient and permits lowering the space requirements on leaf routers. In addition, with absolute XPath expressions, the fanout is guaranteed to never exceed 1. This approach does therefore offer a good tradeoff between the high scalability of the hierarchical strategy and the simplicity and efficiency of the data-sharing strategy.

5 Conclusion

Content-based routing of XML document is an important issue in Web services. In order to scale to large number of clients and high data traffic, Web service applications face the challenging task of efficiently filtering, classifying, and routing incoming XML requests to the appropriate component in the infrastructure. This can be achieved by the combination of highly-efficient filtering algorithms and scalable strategies for parallel filtering. These strategies offer various tradeoffs in terms of throughput, latency, resource consumption, and complexity. They can be combined together to best adapt to the specifics of individual services.

References

[1] C.-Y. Chan, W. Fan, P. Felber, M. Garofalakis, and R. Rastogi. Tree Pattern Aggregation for Scalable XML Data Dissemination. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB 2002)*, Hong Kong, China, August 2002.

[2] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient Filtering of XML Documents with XPath Expressions. In *Proceedings of the 18th International Conference on Data Engineering (ICDE 2002)*, San Jose, CA, February 2002.

[3] The HTRC Group. Data Routing In the Age of Information, October 2001. http://www.htrcgroup.com.

[4] E. Kuznetsov. XML-Aware Networking. *XML Journal*, 3(8), August 2002.

[5] W3C. XML Path Language (XPath) 1.0, November 1999.

[6] W3C. Extensible HyperText Markup Language (XHTML) 1.0, January 2000. http://www.w3.org/TR/xhtml1.

[7] W3C. Extensible Markup Language (XML) 1.0, October 2000. http://www.w3.org/TR/REC-xml.

[8] W3C. Simple Object Access Protocol (SOAP) 1.1, May 2000. http://www.w3.org/TR/SOAP.

[9] W3C. Web Services Description Language (WSDL) 1.2, July 2002. http://www.w3.org/TR/wsdl12.