Scalable Filtering of XML Data for Web Services

Scalable content-based routing architectures for Web applications can handle the growing number of XML messages associated with Web services.

Pascal Felber
Institut EURECOM

Chee-Yong Chan, Minos Garofalakis, and Rajeev Rastogi
Bell Labs, Lucent Technologies

As the Web gains prevalence as an application-to-application communication medium, organizations are deploying more Web service applications to provide standardized, programmatic application functionality over the Internet. Web services use open standards based on the Extensible Markup Language (XML, www.w3.org/TR/REC-xml), such as the Web Services Description Language (WSDL, www.w3.org/TR/wsdl12) for service definition and the simple object access protocol (www.w3.org/TR/SOAP) for service invocation. Users can reach the services using SOAP and construct requests using the Web service's WSDL information. A wide range of domain-specific specifications are also based on XML, such as ebXML for business-to-business interactions and FpML for financial data exchange. Even HTML, arguably the most widely used data format on the Internet, has recently been rewritten as an XML-based specification, Extensible Hypertext Markup Language (XHTML, www.w3.org/TR/xhtml1).

Filtering, classifying, and routing the growing number of XML messages associated with Web services requires scalable mechanisms. Enterprise application servers, for instance, must scale to numerous clients, provide high throughput, and support a variety of XML-based protocols. As Figure 1 illustrates, a Web server (generally associated with a firewall) receives XML data, and one or more XML routers filter it. These routers dispatch XML data, according to its type or content, to the appropriate back-end server, possibly using load balancing, selective multicast, or another routing scheme. XML routers can also act as a sophisticated firewall by filtering out unauthorized or invalid XML messages.[1,2]

Because an XML request's destination depends on its type and content, Web service applications must incorporate highly efficient content-based routing technology. Specialized back-end servers can

 1089-7801/03/$17.00©2003 IEEE Published by the IEEE Computer Society JANUARY • FEBRUARY 2003

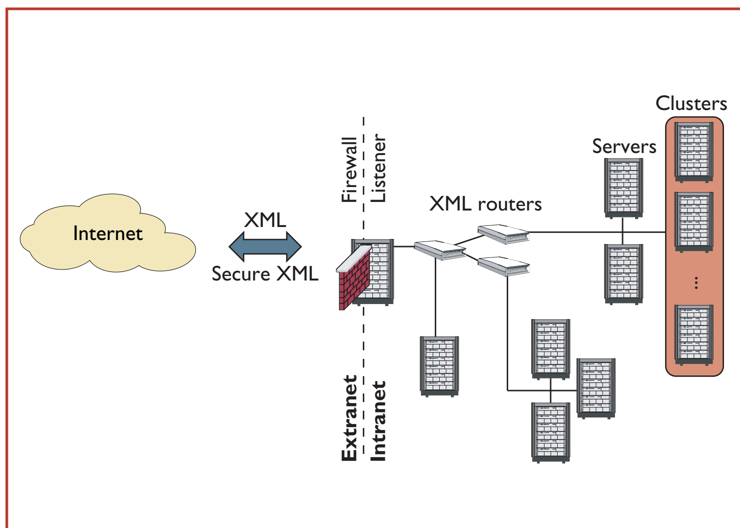

Figure 1. XML routing for enterprise Web servers. Routers filter the XML data received by the Web server and dispatch it to the appropriate back-end servers using various routing schemes.

```
<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>
```

Figure 2. Sample XML code from the SOAP specification. Start and end tags, which are enclosed within < and >, specify the document's structure. Text data, enclosed between tag pairs, express the tag's value or content.

efficiently process only certain types of XML data. A server could be responsible for all SOAP requests, for example, or only for SOAP requests related to stock quotes, or just for specific companies' stock quotes. The resulting services are more scalable and cost-effective than those based on traditional load-balancing schemes, which require each server to process any type of request. Although not specific to Web services, efficient techniques for content-based XML data routing can benefit Web services because of the large amount of XML traffic associated with such services. We have developed techniques for matching XML data to tree-structured filters expressed using the W3C's XML Path Language (XPath, www.w3.org/TR/xpath) and propose a hierarchical XML routing architecture that supports extremely high traffic loads.

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.

Semistructured Data

XML's simple structure is easy for applications to interpret and process. Moreover, being vendor- and platform-neutral and agnostic about content appearance, XML simplifies integration of existing applications and representation of data in various human-readable formats.

XML defines an unambiguous mechanism for constraining structure in a data stream. XML documents can include a type definition (XML schema or document-type definition, DTD), which defines the document structure by describing its legal building blocks. (XML schemas are gaining favor over DTDs, and Web services specifications explicitly disallow DTDs for defining Web service message formats.)

Consider the sample XML document in Figure 2, taken from the SOAP specification, which represents a SOAP request for the current value of a company's stock. Start and end tags specify the document's structure. Start tags contain an optional list of attributes, which are essentially key-value pairs. We refer to text data enclosed between tag pairs, such as between the `symbol` tags in line 13, as the tag's *value* or *content*.

XML documents have a hierarchical structure, as the tree-based representations in Figure 3 show. Attributes (prefixed by the @ symbol) are represented as children of their associated tag; values (shown between quotation marks) are represented as children of their associated attribute or tag.

To distinguish between structural elements — tags and attributes — and the actual data values associated with them, we represent connections between the former with solid lines, and connections between elements and values with dashed lines. Intuitively, an XML document's tags and attributes define its type, while the data associated with its tags and attributes define its value. (Formally, an associated schema specifies the XML document type.) By disconnecting the dashed lines from the XML document's tree rep-

resentation, we obtain its type. Two SOAP requests for the value of different stocks would have the same type but different values.

Tree-Structured Filters

XML's structured, extensible nature allows for a powerful combination of type-based and value-based content filtering. To that end, XML filters should be able to express both data type and value constraints. The simplicity and standardization of the XPath addressing language makes it widely used for that purpose.

XPath treats an XML document as a node tree and lets applications specify and select parts of this tree. An XPath expression contains one or more location steps, which are separated by slashes. A location step designates an element name followed by zero or more predicates between brackets. Predicates use basic comparison operators (=, <, =, >, and =), and are generally specified as constraints on the presence of structural elements or on the values of XML documents. XPath also allows wildcard (*) and ancestor/descendant (//) operators, which match exactly one and an arbitrarily long sequence of element names, respectively. In a data-filtering context, an XML document matches an XPath expression when the expression's evaluation yields a nonnull object.

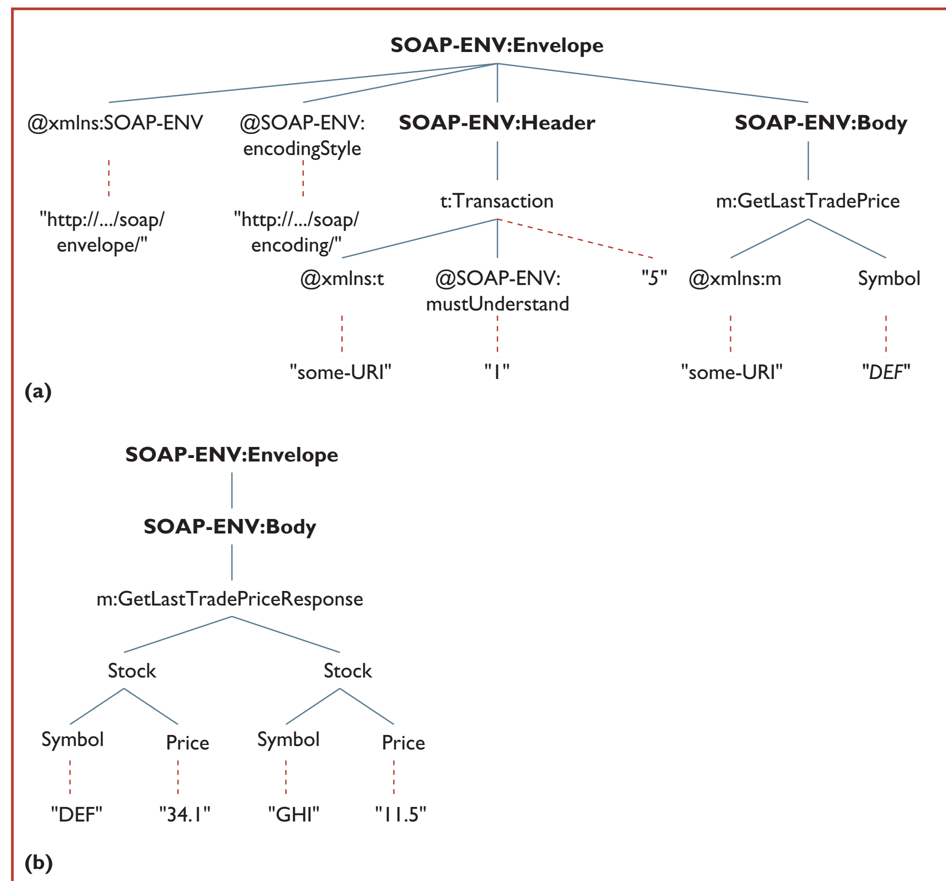

Figure 3. XML trees for sample SOAP messages. (a) Representation of the sample code in Figure 1 without attributes; (b) a SOAP message with multiple response values.

Figure 4 (next page) gives several sample XPath expressions. Figure 4a 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). Figure 4b adds constraints to the Symbol node's value to make it equal "`DEF`". `Text()` selects the text node below the current node. Figure 4c designates SOAP messages that have a `Symbol` node with value "`DEF`" at least two levels inside the message (below the `Body` node). This expression specifies an absolute path from the XML document's root. All three expressions match the XML document in Figure 4a.

Attributes in XPath expressions are specified like tag nodes, but are prefixed with @. Figure 4d designates documents that have a `Transaction` node with a `SOAP-ENV:mustUnderstand` attribute, and Figure 4e further mandates that this attribute have the value 1.

XPath can also 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` with a price higher than 15. Figure 4f expresses the first constraint, and Figure 4g expresses the second. Simply combining these two XPath expressions is not sufficient, however, to obtain the desired filter: the document in Figure 4d contains both expressions, but the price that matches the second expression is not for `GHI`. The filter must further constrain the matching `Symbol` and `Price` nodes to share a `Stock` parent node. We achieve such structural constraints using tree-structured expressions, which we express in XPath by defining multiple predicates on the same node. Figure 4h is one possible embodiment of the desired filter.

```
(a)    //m:GetLastTradePrice/Symbol
(b)    //m:GetLastTradePrice/Symbol[text()=“DEF”]
(c)    /SOAP-ENV:Envelope/SOAP-
          ENV:Body/*//Symbol[text()=“DEF”]
(d)    //t:Transaction/@SOAP-ENV:mustUnderstand
(e)    //t:Transaction[@SOAP-ENV:mustUnderstand=1]
(f)    //Stock/Symbol[text()=“GHI”]
(g)    //Stock/Price[text()>15]
(h)    //Stock[Symbol/text()=“GHI”][Price/text()>15]
```

Figure 4. Sample XPath expressions. Attributes are signified by an @ symbol. Comparison operators (=, <, =, >, and =) indicate predicates, which specify constraints on XML document values. Wildcard (*) operators match one element name while ancestor/descendant (//) operators match an arbitrarily long sequence of element names.

Scalable Architectures for Web Services

The combination of semistructured data and tree-structured filters offers a more flexible and expressive framework for content-based routing than 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.

Efficient Filtering of XML Data

Our XTrie index structure supports the efficient filtering of XML documents based on XPath expressions. It offers several novel features that make it especially attractive for Web service applications with high scalability and performance requirements:

- It supports effective filtering based on complex, tree-structured XPath expressions (as opposed to simple, single-path specifications).
- The XTrie structure and algorithms support both ordered and unordered matching of XML data.
- By indexing on sequences of elements organized in a trie structure and using a sophisticated matching algorithm, XTrie both reduces unnecessary index probes and avoids redundant matchings, thereby providing extremely efficient filtering.

To construct an XTrie for a given set of XPath expressions, we first decompose each expression into a minimal number of substrings, where a substring is a nonempty sequence of elements, separated by a parent/child operator, that can be prefixed by an ancestor/descendant operator and wildcards. We then organize the decomposed substrings using a sophisticated trie structure and an auxiliary table. The trie allows both space-efficient indexing and time-efficient retrieval of XPath expressions, while the table stores additional information used for detecting valid matches.

The trie is a rooted tree, 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 points to other nodes and to rows in the auxiliary table. The table contains one row for each substring of each indexed XPath expression. In each row, a set of values describes the positional and structural constraints of the associated substring. Figure 5 shows an XTrie index for the sample XPath expressions in Figure 4.

The XTrie matching algorithm acts on an incoming XML document as follows. First, the event-based simple API for XML (SAX, www.saxproject.org) parser parses the document, reporting occurrences of XML elements (start tags, text, and so on) to the matcher. The matching algorithm attempts to map sequences of start tags, attributes, and text values to trie paths by following the trie's edges. For each matching substring it detects, the algorithm uses the auxiliary table to verify the substring's positional constraints with respect to its previously matched parent and sibling substrings, as well as any associated predicates. A data structure that stores the occurrence, depth, and scope of previously encountered substrings maintains information about partially matched XPath expressions at runtime. When end tags are parsed, the matching algorithm updates the runtime information to invalidate out-of-scope substring matches.

An XPath expression is completely matched when the algorithm has paired all substrings with their associated constraints and has validated their predicates. Using the event-based SAX parser, which does not require an in-memory representation of the input document for matching, XTrie can also filter streaming XML data. We provide an exhaustive description of the XTrie algorithms elsewhere.[3]

Parallel XTrie

Because they must handle many XPath expressions and process many requests, large Web service applications usually impose very demanding performance requirements. One way to improve the XTrie's scalability is to parallelize its processing so that multiple XML routers can share the time- and space-consuming task of filtering data. We can easily achieve parallelization using a cluster of XML routers organized according to one of

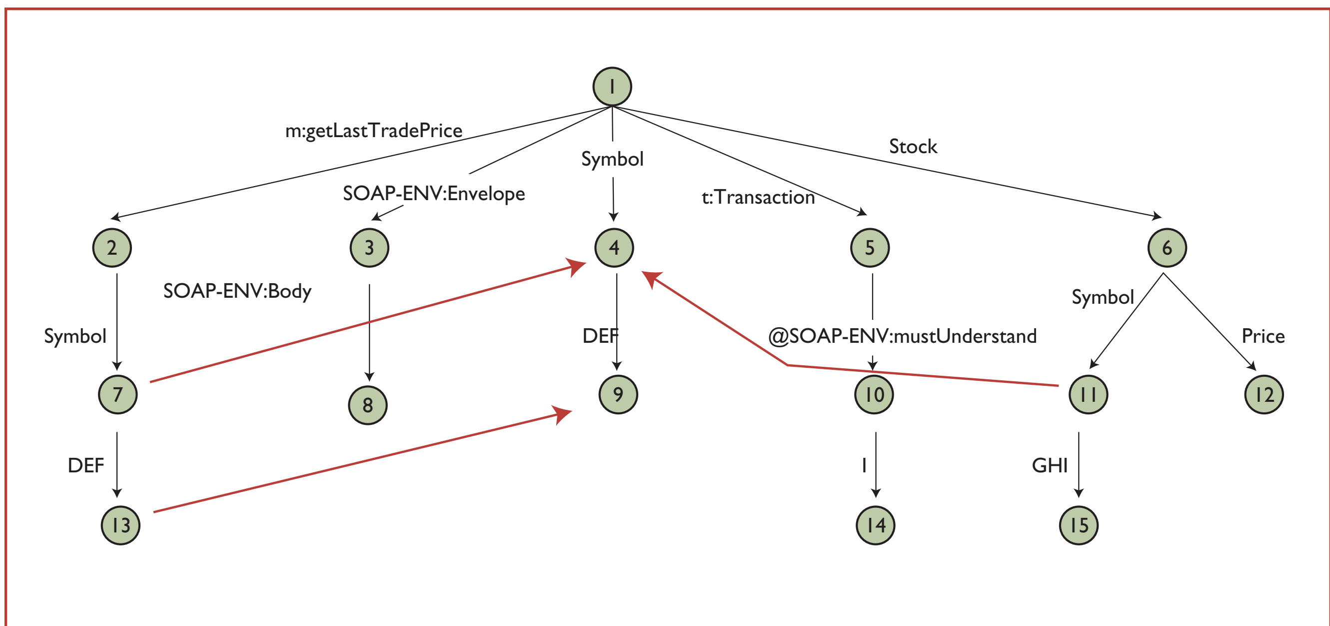

Figure 5. XTrie for the sample XPath expressions in Figure 4. Each edge in the trie is labeled with an element name; the auxiliary table is not shown.

two simple strategies.

- *Data-sharing strategy.* Each XML router in the cluster manages the complete set of XPath expressions. A load balancer dispatches each XML document to only one router according to a load-balancing strategy (Figure 6a).
- *Filter-sharing strategy.* XML routers share XPath expressions equally, so exactly one router manages each distinct expression (Figure 6b). All routers filter incoming XML documents.

We designed these two strategies to optimize different scalability requirements. The first strategy maximizes the filtering throughput by increasing the number of documents that can be processed concurrently. 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.

Hierarchical XTrie

One way to achieve both reasonable filtering throughput and reasonable filtering response time is to combine the strengths of the data- and filter-sharing strategies into a hybrid strategy that organizes the cluster of XML routers into a hierarchical configuration. The network dispatcher sends each incoming XML document to the root router, performing coarse filtering to decide which subset of child routers it will send the document to for more refined filtering. This top-down propagation and XML document filtering continues from one level to the next until the document reaches the leaf level, where a subset of leaf routers filter it to decide which target back-end servers to dispatch the document to.

Figure 6c (next page) is an example of the hierarchically organized XTrie. In the hierarchy, each router manages a set of filters (that is, XPath expressions). We use the filter-sharing strategy to organize leaf routers, so each leaf router manages a disjoint subset of the XPath expression (or *end filter*) workload. 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 its corresponding child router manages. Specifically, each filter *F* in a child router must be covered by some filter in *F'* in its parent router so any XML document that matches *F* will also match *F'*. For instance, the XPath expression `//Symbol` covers the XPath expressions in Figures 4a through 4c, 4f, and 4h. This property guarantees that whenever a document matches an end filter in a leaf router, the internal routers will always route the document to that leaf router. This hierarchical strategy therefore combines the advantages of the filter- and data-sharing strategies: like the former, multiple routers process each document to improve filtering latency; like the latter, routers can process

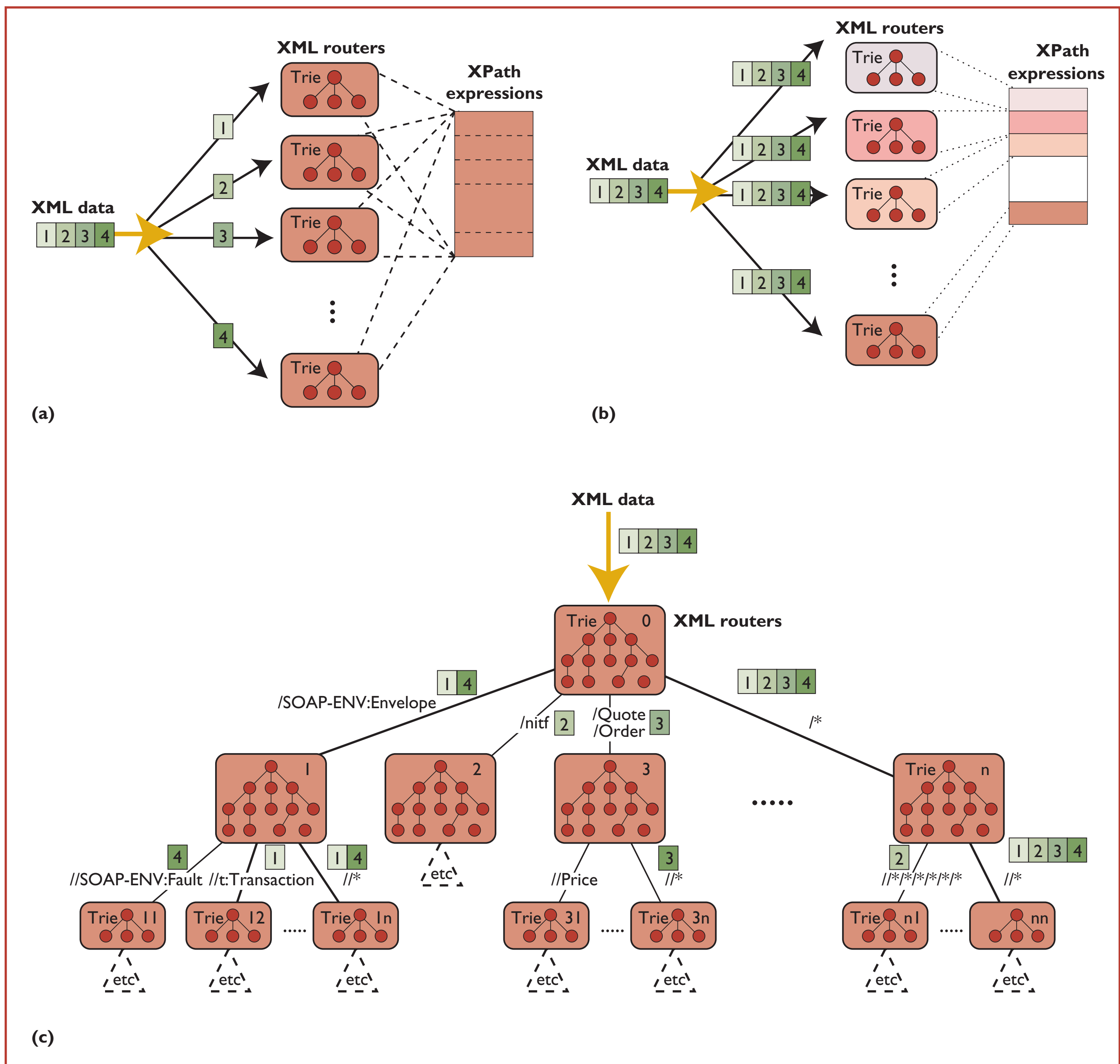

Figure 6. Strategies for parallel filtering of XML documents with XTrie: (a) sharing XML workload, (b) sharing XPath expressions, and (c) hierarchical filtering (highlighted paths are traversed by the sample SOAP message in Figure 2).

multiple documents concurrently to improve filtering throughput.

The main challenge of the hierarchical strategy is clustering the XPath expressions into subsets to be assigned to leaf XML routers, and clustering the collection of XML routers at one level into subsets to be assigned to parent routers at the level above. This clustering must be performed in such a way that

- a small set of intermediate filters installed in the parent router can concisely represent each subset, and
- each XML document is propagated down to only a small subset of leaf routers.

The first condition ensures efficient filtering at internal routers as each router manages only a small set of filters, while the second condition optimizes the filtering throughput by ensuring that only the relevant subset of routers processes each XML document.

Clustering XPath expressions. There are a number of simple methods for clustering 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 with distinct root nodes must have different types (schema).

The first level of the hierarchical XTrie in Figure 6c illustrates this partitioning. The topmost router filters XML documents according to their root tags and forwards them to the appropriate child router subset downstream. For instance, router 1 handles all SOAP requests, while router 3 handles both quote and order requests. The subtree at router *n* contains all XPath expressions that do not have an explicit root tag (that is, they begin with // or /*). Thus, in Figure 6c, the topmost router will always propagate each XML document to router *n*, as well as to any router associated with the document's root tag. More generally, we can partition nonabsolute XPath expressions using schema-specific element names.

Clustering XPath expressions by the XML document type they refer to is effective in practice, but it only permits building a coarse-grained router hierarchy. This might prove inefficient when, for instance, some types of documents occur much more frequently than others. It is therefore desirable to also cluster XPath expressions in a single schema. Router 11 in Figure 6c manages SOAP error messages, for example, and router 12 filters SOAP messages with transaction identifiers. For good results, you can 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 you know the XML schemas in advance. Alternately, you can observe XML data flowing through the routers and gather 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 cover the set of XPath expressions in its child routers. To provide effective coarse-level filtering, the set of intermediate filters needs to satisfy two conflicting requirements. First, it should be small to enable fast filtering. Second, it should provide a tight covering — that is, it should minimize the number of documents matching some intermediate filter in a router but not matching any filters in the router's corresponding child router. In other words, it should minimize unnecessary document forwarding to irrelevant downstream XML routers.

When XPath expressions are clustered according to element names, the construction of the intermediate filters is trivial. When the set of XPath expressions managed by a router is diverse, however, finding a set of intermediate filters is challenging. In earlier work, we proposed an efficient aggregation algorithm that effectively uses document-distribution statistics to compute a precise and compact set of coarse-level XPath expressions for a given set of XPath expressions.[4] We can apply this algorithm to intermediate filter generation. 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.

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 actual document types and numerous tree patterns. We then evaluated the performance of the various architectures for parallelizing XTrie based on our results. Because an XML message's filtering time is several orders of magnitude slower than its actual transmission over the network, we did not account for transmission time. Our study thus shows the asymptotic performance of the parallel XTrie architectures. We conducted our experiments on a 1.5-GHz Intel Pentium 4 machine with 512 MBytes of main memory running Linux.

For our experiments, we selected 10 real-world commercial application document types with 77 to 2727 distinct elements and up to 8512 distinct attributes. Most of these types have recursive structures, which we can nest to produce XML documents with an 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 percent of * and // operators.

We ran experiments with two variants of the XTrie algorithm:

- one optimized for single-path XPath expressions, where each node has at most one child; and
- another optimized for ordered matching of tree-structured XPath expressions.

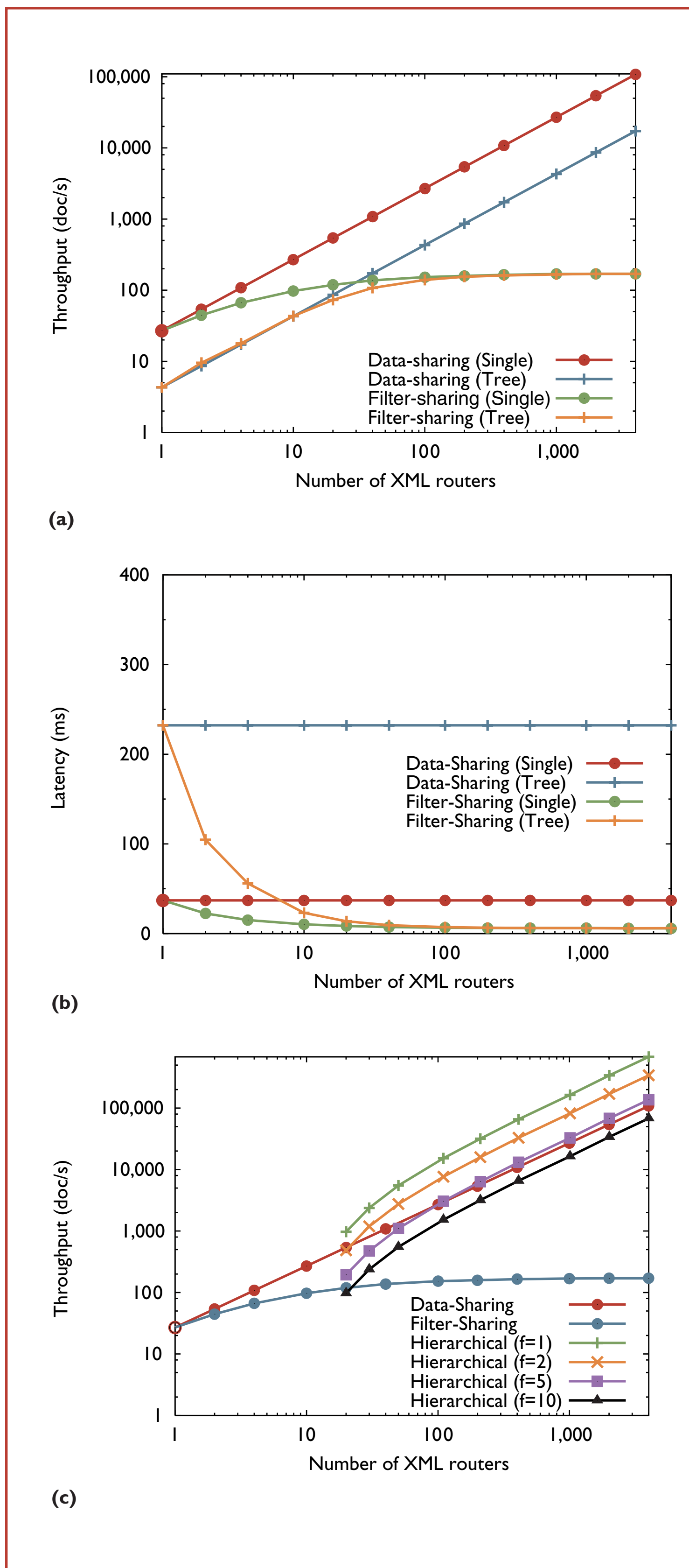

Figure 7. Parallel XTrie filtering performance with 200,000 single-path XPath expressions: (a) throughput, (b) latency, (c) hierarchical filtering throughput (single-path XTrie algorithm).

We compared the scalability of XTrie parallelization for the data-sharing, filter-sharing, and hierarchical strategies, using simulation based on the experimental results of the XTrie algorithms.

Raw XTrie Performance

The performance of the nonparallel 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 elsewhere.[3]

Data-Sharing and Filter-Sharing Strategies

Figure 7a compares the throughput performance of the data-sharing and filter-sharing strategies (with logarithmic scales) for 200,000 XPath expressions. As expected, the data-sharing strategy's throughput scales linearly with the number of XML routers. In contrast, the filter-sharing strategy's throughput increases more gradually with the number of XML routers to 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 how many XPath expressions each router manages).

Figure 7b compares the latency performance of the parallel XTrie strategies 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 because only one of the routers filters each incoming document. For the filter-sharing strategy, on the other hand, 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 for a Web service application depends on the performance objective.

Hierarchical Strategy

We evaluated the performance of the hierarchical strategy with absolute XPath expressions (that is, relative to the root node) and a clustering scheme in which 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. Because all the input XPath expressions are absolute, an XML document is routed to at most one subtree during first-level routing.

In practice, two major factors prevent hierarchical filtering from being optimal:

- The average number of leaf routers ultimately reached by each XML document — its *fanout* — is generally greater than 1 and reduces parallelism.
- Contention on some leaf routers that are traversed more often than others might exist.

To account for these factors, we evaluated the performance of the hierarchical configuration with different values for the average fanout f. Figure 7c shows the results. 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, $f = 1$, scales the best; however, the data-sharing strategy is preferable for fanouts higher than 5.

A promising alternative is to combine the hierarchical and data-sharing strategies. We can use a one-level routing hierarchy that partitions the XPath expressions based on their root nodes, and then use the data-sharing strategy to load-balance XML documents to clusters of XML routers dimensioned according to the document distribution. As we mentioned previously, the first-level filtering phase is extremely efficient and lets us lower leaf router space requirements. In addition, with absolute XPath expressions, the fanout will never exceed 1. This approach therefore offers a good trade-off between the high scalability of the hierarchical strategy and the simplicity and efficiency of the data-sharing strategy.

Conclusion

Our XML filtering techniques can facilitate the design of highly scalable Web services. Several important issues, however, remain open for further research. In particular, the configuration and clustering of routers is a crucial issue with significant impact on routing performance. We are also working to extend our techniques to better adapt to fluctuating traffic loads and data distributions.

References

1. The HTRC Group, "Data Routing in the Age of Information," Oct. 2001. http://www.htrcgroup.com.
2. E. Kuznetsov, "XML-Aware Networking," *XML J.*, vol. 3, no. 8, Aug. 2002, pp. 22-23.
3. C.-Y. Chan et al., "Efficient Filtering of XML Documents with XPath Expressions," *Proc. 18th Int'l Conf. Data Eng.* (ICDE 2002), IEEE CS Press, 2002, pp. 235-244.
4. C.-Y. Chan et al., "Tree Pattern Aggregation for Scalable XML Data Dissemination," *Proc. 28th Int'l Conf. Very Large Data Bases* (VLDB 2002), Morgan Kaufmann, 2002, pp. 826-837

Pascal Felber is an assistant professor at Institute EURECOM, France, a graduate institute specialized in Telecommunications. He has MSc and PhD degrees from the Computer Science Department of the Swiss Federal Institute of Technology (Lausanne). His current research interests include dependable and distributed systems, distributed data management, and object-based systems. He is a member of the ACM and the IEEE Computer Society. Contact him at pascal.felber@eurecom.fr.

Chee-Yong Chan is a member of technical staff at the Information Sciences Research Center of Bell Labs, Lucent Technologies. He has an MSc from the Computer Science Department of the National University of Singapore, and a PhD from the University of Wisconsin-Madison Computer Science Department. His current research interests include XML databases, indexing techniques, and data-warehousing systems. He is a member of the ACM and the IEEE Computer Society. Contact him at cychan@research.bell-labs.com.

Minos Garofalakis is a member of technical staff at the Information Sciences Research Center of Bell Labs, Lucent Technologies. He has a BSc from the Computer Engineering and Informatics Department of the University of Patras, and MSc and PhD degrees in computer science from the University of Wisconsin-Madison. His current research interests lie in the areas of data streaming, approximate query processing, data mining, network management, and XML databases. He is a member of the ACM and the IEEE. Contact him at minos@research.bell-labs.com.

Rajeev Rastogi is the director of the Internet Management Research Department at Bell Laboratories, Lucent Technologies. He has a BSc in computer science from the Indian Institute of Technology, Bombay, and MSc and PhD degrees in computer science from the University of Texas, Austin. His research interests include database systems, network management, storage systems, and knowledge discovery. His most recent research has focused on network topology discovery, monitoring, configuration and provisioning, data mining, and high-performance transaction systems. Contact him at rastogi@research.bell-labs.com.