BFilter – A XML Message Filtering and Matching Approach in Publish/Subscribe Systems

Liang Dai
School of Computer Science
Carleton University, Ottawa, Ontario, Canada
liang_dai@yahoo.com

Chung-Horng Lung, Shikharesh Majumdar
Department of Systems and Computer Engineering
Carleton University, Ottawa, Ontario, Canada
{ehlung, majumdar}@sce.carleton.ca

Abstract - In publish/subscribe systems, XML message filtering performed at application layer is an important operation for XML message multicast. As a specific case of content-based multicast in application layer, XML message multicast depends on the data filtering and matching processes and the forwarding and routing schemes. As the XML data emerges in transition, XML message filtering and matching becomes more and more desirable. BFilter, proposed in this paper, conducts the XML message filtering and matching by leveraging branch points in both the XML document and user query. It evaluates user queries that use backward matching branch points to delay further matching processes until branch points match in the XML document and user query. In this way, XML message filtering can be performed more efficiently as the probability of mismatching is reduced. A number of experiments have been conducted and the results demonstrate that BFilter has better performance than the well-known YFilter for complex queries.

Keywords - XML; XML message filtering and matching; pub/sub systems

1. INTRODUCTION

In publish/subscribe (pub/sub) systems or Web services, application layer multicast is widely used for data dissemination to subscribers. In pub/sub systems, a subscriber registers a subscription to the pub/sub service and receives published messages that match the subscription. Intuitively, the source (publishers) can allow their subscribers to retain whatever they want, and send all the data to all subscribers. This approach is definitely not efficient because there are too many duplicated data packets.

Generally speaking, there are two ways to carry out multicast in the context of pub/sub systems [2,4,7,8,9,11,13,17,22,23]. The first is to find the subscriber by using the subscription information, and then send appropriate data to subscribers. Data matching can be performed either at the source or at some centralized brokers. The second method is to perform data matching on the fly. In this way, the source simply pushes the data into the network that has a multicast tree composed of routers or brokers. The routers or brokers on the tree have filters to dispatch proper subsets of data to their children. The children in turn perform data matching and dispatching and forward the matched data to their children. This continues until the filtered data reaches the subscribers.

The first approach described above may use keyword-based multicast [2,11,15,16,18,19,21] or distributed hash table-based multicast. Distributed hash table-based multicast uses hash functions to assign keys to subscribers by using their subscriptions [5]. These methods are efficient in terms of delivery speed. However, the keyword-based approach is less expressive because the subscriptions contain only keywords. The distributed hash table approach is not content-aware. In these methods, data matching is based on keywords but not the content. The second approach delivers data according to the content. The subscription description is used to perform the matching. The subscription can be presented either in an n-tuple containing n information spaces, or in XPath expressions [1,6,13,14]. An XPath expression is used for addressing portions of a XML file. XPath is more expressive than n-tuple.

A XML file is a tree-based structure for describing information. As a XML file is structured, it naturally applies filters in the hierarchy to perform data matching and delivery. XML-based multicast can properly match and deliver messages to subscribers. However, because it is difficult to index and identify the elements in the XML file, the filtering process in each node is time consuming. Hence, the performance of XML-based multicast depends heavily on the approach used to process the XML message.

Several approaches for XML filtering have been reported in the literature, see Section 2 for details. One common limitation of those approaches for complex queries that have nested paths is that complex queries have to be decomposed into sub-queries and a post-processing task is needed. As a result, the filtering process becomes inefficient. This paper proposes a novel XML message filtering algorithm—BFilter. (B represents branch points.) BFilter realizes the tree structure in both XML documents and user queries with nested paths. It conducts the XML message filtering and matching process by identifying branch points in both XML documents and user queries. The evaluation of user queries uses backward matching branch points to delay further matching, so that the probability of a mismatch is reduced and XML message filtering can be performed more efficiently.

The rest of the paper is organized as follows. Section 2 presents the background. Section 3 discusses the backward matching branch point algorithm. Section 4 demonstrates the some experimental results. Finally, section 5 is the summary.

2. BACKGROUND AND RELATED WORK

There are two important operations performed in a pub/sub system: XML message filtering and multicast. This paper focuses on techniques for XML message filtering.

XFilter [1] is based on deterministic finite automata, which stores user queries and handles each query individually. It is capable of handling XPath relationship notations, such as ancestor/descendant (represented by ‘//’ in XPath) as well as
wildcard ‘*’. Because it stores queries separately, the same segments of a different query cannot share storage space and must be matched individually.

YFilter [12] overcomes the disadvantage of XFilter by using nondeterministic finite automata (NFA) to emphasize prefix shifting. However, the ancestor-descendant relationship introduces more matching states, which may result in the number of active states increasing exponentially [20]. Post-processing is required for YFilter. To deal with queries with nested paths (complex queries), YFilter decomposes them into simple queries and matches them separately. For example, a complex query /a/*/c/*/c will be split into /a/*/c (Q1) and /a/*/c (Q2). After NFA execution is finished for the entire document, post-processing starts to verify if Q1 and Q2 are both matched.

In contrast to YFilter, AFilter emphasizes suffix commonality to reduce mismatches [20]. The objective of AFilter is to overcome the disadvantage of YFilter by constructing user queries as a directed graph (AxisView). AFilter uses a triggering mechanism to delay the matching process until a trigger condition is met. Each node in the directed graph corresponds to a label and each edge corresponds to a set of axis tests. Each edge is annotated with a set of axis assertions for verification. Unlike YFilter, which uses a stack to store matched active states, AFilter uses a set of named stacks to store matched states. The states are called objects because they also maintain pointers that allow tracing of previous matching states. Therefore, the state stacks actually store the matching paths. AFilter implements these data structures to backtrack the matching path whenever an end of query is triggered during the process of matching. AFilter does not explicitly deal with queries with nested paths.

GFilter [10] focuses on optimizing the path matching performance via a bottom-up approach, benefiting from the heuristics that the probability of mismatching at the leaf is higher than at higher levels. However, the post-processing is needed for handling complex queries.

3. BFilter—BACKWARD BRANCH POINT MATCHING

BFilter effectively deals with user queries with complex queries by leveraging branch points in both XML document and user query. BFilter not only matches queries backwards, but also matches branch points backwards. The matching of branch points is preconditioned for further matching of a complex query. Since BFilter treats a complex query as a whole, no post-processing is needed for complex queries.

3.1 Technique Overview

In BFilter, a complex query is treated as a unit without being decomposed. BFilter realizes the tree structure of XML documents and user queries. A tree is composed of three parts: branch points that include the root; branches connecting two branch points; and branches that have only one end attached to a branch point while the other end is free. A branch that connects two branch points called a Transit Branch; other branches are called Tangling Branches as shown in Figure 1.

The matching of BFilter process starts only when both a document branch point and a query branch point have been matched. Only when the branch point is matched will its tangling branches and the transit branches (if they exist) are checked. In other words, the matching of branches is delayed until the branch point they attach to is matched.

Given any query Q with nested path and a XML document, let the probability of matching of Q be P(Q). Since YFilter matches Q from the front to the end of Q, we have:

$$P(Q) = P(front) \times P(rest | front),$$

where P(front) is the probability of matching of the front part of Q and P(rest | front) is the probability of matching of the rest of Q when the front is matched. Similarly, AFilter matches Q from the end to the front of Q:

$$P(Q) = P(end) \times P(rest | end),$$

where P(end) is the probability of matching of the end part of Q, and P(rest | end) is the probability of matching of the rest of Q when the end is matched.

Because BFilter matches Q from the last branch point to the front of Q, we have:

$$P(Q) = P(last \ branch) \times P(rest | last \ branch),$$

where P(last branch) is the probability of matching of the last branch point of Q and P(rest | last branch) is the probability of matching of the rest of Q when the last branch is matched.

Liang [24] demonstrated that P(rest | last branch) > P(rest | end) > P(rest | front). In real messages, a tag name has a higher probability of appearance at a higher level than at a lower level in a query. In other words, the probability of a match of the front part of a query is higher than that of its end part, which indicates that a match for the front part of a query may not lead to successful match for the end part. This observation motivates a backward matching (bottom-up) approach used in XML filters including AFilter, GFilter as well as BFilter. It implies that, for a particular query, if the match of its last element(s) is successful during backward matching, the probability of a match of its remaining part is higher than the case in which the match of its front element(s) is successful during forward matching.

In other words, the probability of mismatching of branch point backward matching is less than that in backward matching (proposed in BFilter), and the probability of mismatching of backward matching (adopted by AFilter and GFilter) is less than that in forward matching (used in YFilter). In other words, the probability that mismatching will occur in BFilter is expected to be the lowest of the three. Mismatching is a scenario where the matching fails after the first step(s).
succeeds. Mismatching causes the filter to spend time on evaluating queries that will ultimately fail. The lower the probability of mismatching, the greater the likelihood that the processing of un-matched queries will be stopped earlier in the matching process. Reducing the mismatching probability increases the efficiency of filtering.

3.2 Query Representation

BFilter represents a complex query using sub-queries that are separated by branch points. Figure 2 shows the representation of a query in BFilter. Each sub-query is rooted at a branch point and has one or more paths. A sub-query is named as Q\{m,n\}, where m is the index of the complex query the sub-query belongs to, and n is the index of the sub-query which starts from 0. A sub-query has two pointers, one pointing to its parent and another to its child.

In Figure 2, Q1 is a simple query; its root is a and it has a single path /a/b/c. Q2 is a complex query that has one branch point (the root), and it has root a and two paths /a/d and /a/b/c. Q3 is a complex query with three branch points, and is represented by three sub-queries, Q\{3,0\}, Q\{3.1\} and Q\{3,2\}.

3.3 Algorithm

Figure 3 shows how to handle the start tag event during filtering process. Figure 4 depicts the procedure for matching branches when a match of branch point is found.

Figure 5 presents the filtering algorithm. The first if statement tests if a start element or tag is read in. When a start tag is read in, the element is pushed onto the runTimeBranch stack. If a branch point is also detected, the top of the runTimeBranch stack will be copied to the documentBranches Stack and all the queries associated with the branch point will be pushed onto the queryBranches stack.

When the end tag of an element is read in, the element is popped out from the runTimeBranch Stack. At the same time, both documentBranches stack and queryBranches stack are checked to see if a branch point that corresponds to this element exists in both stacks. If it does, the matching process

\[
\text{Procedure MatchingBranches()}
\]

\text{For each matched query q, do}
\text{If q's child is not matched}
\text{\quad // no matching is needed}
\text{\quad Set q as not matched;}
\text{\quad Else}
\text{\quad Process matching on all branches of q.}
\text{\quad If q is failed in matching}
\text{\quad \quad // reset states of all sub-queries}
\text{\quad \quad Set subqueries as unmatched;}
\text{\quad \quad End if;}
\text{\quad End if;}
\text{End for;}
\text{End Procedure;}

\[
\text{Procedure HandlingStartTag(Document tag e)}
\]

\text{// get current state in glnIndex for e}
\text{Send e to glnIndex for states lookup;}
\text{For each current state c}
\text{If c is branch point for query q in Q}
\text{Push (c,e, q) onto queryBranches;}
\text{If c is accepting state for query q in Q}
\text{If q has no predicates on this path}
\text{Make the path as matched,}
\text{Else}
\text{// delay the matching until}
\text{// the branch point is matched.}
\text{Save branch in the node;}
\text{End if;}
\text{End if;}
\text{End for;}
\text{End Procedure;}

\[
\text{Figure 2. Example Query}
\]

\[
\text{Figure 3. Handling Start Tag}
\]

\[
\text{Figure 4. Matching branches}
\]

\[
\text{Figure 5. Filtering Algorithm}
\]
starts for the sub-queries that have this branch point; otherwise the matching process is not performed. The matching process includes finding matches for all the sub-queries’ branches and evaluating the predicates on the branches. Finally if the corresponding branch point is stored in documentBranches stack and queryBranches stack, it is popped out from these two stacks.

Figure 6 demonstrates how BFilter algorithm works with a concrete example. In the example, four queries, Q1 to Q4, have been aggregated. The queries, their sub-queried and associated paths are:

- Q1: /a/b/e
  Q{1,0}root=a, p0=a/b/e
- Q2: /a/b/c/e
  Q{2,0}root=a, p0=a/b
  Q{2,1}root=b, p0=/b/e, p1=/b/c
- Q3: /a/c/e
  Q{3,0}root=a, p0=/a//c
  Q{3,1}root=c, p0=/c/d, p1=/c/e
- Q4: /a/*/b/c
  Q{4,0}root=a, p0=/a/*
  Q{4,1}root=*, p0=/*/c, p1=/*/b

Figure 6(a) depicts the execution in the query index tree and the changes of the status of the Runtime Stack which is used to hold active states during matching process. Figure 6(b) shows the changes of the statuses of documentBranches and queryBranches. The incoming XML document stream or message is a/b/c/d/e/e/b/a, where underscored letters represent standard XML end tags.

At the beginning the Runtime Stack is empty. When the first element, a, is read in, state 2 is reached in the QueryIndex Tree and is pushed onto the Runtime Stack. State 2 matches four branch points associated with element a of the queries previously aggregated in the tree. These four branch points are the first sub-queries of Q1, Q2, Q3 and Q4 and are represented as Q{1,0}, Q{2,0}, Q{3,0} and Q{4,0}, respectively. These branch points are wrapped as a unit which is associated with the element a so that they can be found when the end tag of element a is read in. Because the first element is also the root of the XML document, a node associated with element a is pushed onto the documentBranches Stack.

When element b arrives, three states are active. First, from state 2 to state 3, state 3 becomes active. Second, because state 6 is the next state of state 2 that requires no input, state 6 is also active. Finally, the transition of "*" from state 2 to state 10 matches any tag name, so state 10 is active too. Thus, states 3, 6 and 10 are pushed onto the Runtime Stack. When element c arrives, top states 5, 7, 12 and 6 are deleted from the Runtime Stack. Because element c has a corresponding branch point in queryBranches Stack but not in documentBranches Stack, the corresponding branch point is popped out from queryBranches and Q{3,1} is marked as unmatched. No matching will be processed for the sub-query Q{3,1}, since element c is not a branch point in the document.

Again, the process is repeated until g is read in. When it happens, BFilter finds a match of branch points in the stacks. The four branch points, namely Q{1,0}, Q{2,0}, Q{3,0} and Q{4,0}, match branch point a in the document. Because the child query Q{4,0} is unmatched, Q{4,0} is simply marked as unmatched without undergoing the matching process. Q{1,0} is simple because it has only one path and is marked as matched. Because the child query of Q{2,0} is matched, Q{2,0} is marked as matched after the match of its transit branch succeeds. Similarly the Q{3,0} is marked as unmatched because its child is unmatched. At the end of this process, BFilter checks whether a query is matched or not by looking at its root in the sub-query representation. Thus Q1 and Q2 are matched in this example.

Figure 6. Execution in BFilter

Note that the matching process only starts when a root of a sub-query is popped out from the queryBranches Stack.
Whenever a match is determined for a sub-query, no matter whether the result is matched or not, all matched candidates of the sub-query are deleted. If the result is unmatched, all the descendants of the sub-query are marked as unmatched.

This example demonstrates the difference between YFilter and BFilter. BFilter matches queries backwards and the matching process only starts when branch points match in both the document and the queries. For queries Q1, Q2, Q3 and Q4 in the example, YFilter will execute the matching process at each accepting state for the corresponding queries that are decomposed from the four queries.

In particular, in YFilter, Q3 will be decomposed into two simple queries Q3.1 (\(/a/*/c\)) and Q3.2 (\(/a///e\)) and Q4 will be decomposed into two simple queries Q4.1 (\(/a/*/c)\) and Q4.2 (\(/a/*/b)\). From the QueryIndex Tree we can see that states 9 and 12 will be the accepting states for Q3.1 and Q4.1, respectively. Thus the matching process will occur when the two states are reached in YFilter. However, no matching processing is needed for Q3 and Q4 in BFilter, since the branch points are not matched (a and b in the document stream, a and e in Q3, and a and b in Q4). In the case of a document having more content after the end tag of element b is read; for instance, the example document stream, \(a/b/c/d/d/e/e/b/a\), has more content between b and a, and if this extra part matches Q3.2 or Q4.2, YFilter not only processes matching for Q3.2 or Q4.2, but also conducts post-processing to verify these decomposed queries and will eventually discover that Q3.1 and Q3.2, as well as Q4.1 and Q4.2, are not matched in the same place in the document. The post-processing in cases like these for YFilter is unnecessary and the time is wasted.

Unlike YFilter, BFilter will clean up all candidates for the current sub-queries as well as their descendants when the corresponding branch point in the document is read over. BFilter only keeps matching candidates for the current portion of the XML document. Thus, if a sub-query and its descendants are matched, it needs not do anything when the new portion of the document arrives. Otherwise, the matching of branch point restarts from the root of the last sub-queries in the new portion. From this example we can see that BFilter is more efficient to deal with complex queries than YFilter.

4. EXPERIMENTAL RESULTS

As discussed earlier, BFilter is built on top of YFilter to reuse existing components. The BFilter solution was implemented using Java 1.6 and run on an AMD 1.6 GHz processor with 1.0 Gb of main memory running on Vista. The data type used in the experiments was the same as in YFilter: News Industry Text Format [12]. ToXGene [3] was used to generate the XML document. A total of 10 documents were used in this experiment.

The query generator in the YFilter test suite [17] was used to generate the queries. Parameters, such as query depth (L), number of branches, number of predicates (P) and probability of ‘//’ and wildcard ‘*’ were varied in various experiments.

4.1 The effect of the number of nested paths

The number of nested paths (NP) is varied from 0 to 5. Figure 7(a) presents the results for two groups, each having identical parameters for YFilter and BFilter. Figure 7(b) presents the results for another two groups. In every group of comparison, BFilter performs better than YFilter except when the number of nested paths equals zero, i.e., when the queries are simple. This is because the current implementation of BFilter reuses the components in YFilter and reconstructs the classes on top of YFilter components. It has no performance gain to offset the overhead in the case of a simple query.

In YFilter, when the number of nested paths increases the number of simple queries decomposed from complex queries also increases. Therefore, YFilter’s filtering cost tends to increase because it needs to do more work during post-processing. In the case of BFilter, because the number of nested paths increases, the probability of branch point matching will decrease. Thus, the processing for matching will be reduced in BFilter, and BFilter’s filtering cost will decrease.

4.2 The effect of probability of ‘//’

In this experiment, the probability of ‘//’ that appears in the queries is varied from 0 to 0.4. Figure 8(a) presents the results for two groups, each having a YFilter and a BFilter. Figure 8(b) presents the results for another two groups. The results also reveal that BFilter is faster than YFilter when the number of nested paths is not zero. The number of nested paths is zero, YFilter is faster than BFilter except when the probability of ‘//’ is less than 0.1.

As mentioned above, BFilter reuses the query index tree of YFilter. It suffers the same overhead as explained in YFilter due to the large number of active states produced by ‘//’. This can be seen from the figure: the filtering cost of both YFilter and BFilter increases when the probability of ‘//’ increases.

4.3 The effect of probability of ‘*’

In this experiment, the probability of ‘*’ that appears in the queries is varied from 0 to 0.4. The effect of ‘*’ on the NFA is similar to that of ‘//’. The results also demonstrate that when the percentage of ‘*’ increases, the filtering cost of both
YFilter and BFilter increases. This is because the wildcard matches any incoming tag from a XML document so that both YFilter and BFilter need to do more work due to the increase of active states. More detailed results can be found in [24].

5. CONCLUSIONS AND DISSCUSSION

XML has become the standard for data exchange. Processing XML stream data has received considerable attention as the filtering problem is critical to XML pub/sub systems. We proposed a novel XML filtering and matching algorithm, BFilter, which performs XML message filtering and matching by leveraging branch points in both XML documents and user queries.

Unlike YFilter, BFilter matches user queries with branches without having to decompose them. BFilter treats a complex query as a whole, and hence no post-processing is needed. In comparison to AFilter and GFilter, BFilter not only processes matching backward but also utilizes branch point matching as a precondition for further steps. The matching of query branches is delayed until the branch point they attach to is matched. In this way, BFilter has a high probability of detecting mismatching early in the matching process; hence, BFilter can reduce the time for unnecessary matching process.

The measurement results from the current implementation show that BFilter performs better than YFilter while handling complex queries. Because BFilter is built on the top of YFilter, it should have similar behaviour as YFilter does. The gain of BFilter is not from the implementation but from the filtering algorithm – backward branch point matching.

ACKNOWLEDGEMENTS

The authors wish to acknowledge Alcatel-Lucent and Ontario Centres of Excellence for their support.

REFERENCES


