Fast Forwarding for Content-Based Networking

Antonio Carzaniga Alexander L. Wolf

Software Engineering Research Laboratory
Department of Computer Science
University of Colorado
Boulder, Colorado 80309-0430 USA
{carzanig,alw}@cs.colorado.edu

University of Colorado
Department of Computer Science
Technical Report CU-CS-922-01 November 2001

revised
September 2002

© 2002 Antonio Carzaniga and Alexander L. Wolf

Abstract

This paper presents an algorithm for content-based forwarding, an essential function in content-based networking. Unlike in traditional address-based unicast or multicast networks, where messages are given explicit destination addresses, the movement of messages through a content-based network is driven by predicates applied to the content of the messages. Forwarding in such a network amounts to evaluating the predicates stored in a router's forwarding table in order to decide to which neighbor routers the message should be sent. We are interested in finding a forwarding algorithm that can make this decision as quickly as possible in situations where there are large numbers of predicates and high volumes of messages. We present such an algorithm and give the results of studies evaluating its performance.

1 Introduction

A content-based network is a novel communication infrastructure in which the flow of messages through the network is driven by the content of the messages, rather than by explicit addresses assigned by senders and attached to the messages [7, 10]. In a content-based network, receivers declare their interests to the network by means of *filters*, while senders simply inject messages into the network at the periphery. The network is responsible for delivering to each receiver any and all messages matching the filter declared by that receiver. An ideal application for a content-based network is a publish/subscribe event notification service [2, 3, 6, 13, 14], where a filter represents a subscription and a message represents a published event.

As in traditional address-based networks, the delivery function is performed incrementally by passing messages between intermediate nodes in the network. We say that messages flow from *upstream* nodes to *downstream* nodes until delivered. The delivery function consists of two interrelated subfunctions: *routing* and *forwarding*. Routing amounts to establishing flow paths through the network by compiling and positioning local forwarding tables at each node. A forwarding table contains the information necessary for a node to decide to which neighbor node or nodes a given message should be sent; the processing of a message at a node is the forwarding subfunction. Taken together, the forwarding performed at the nodes causes messages to be routed through the network. It is unfortunate that intermediate nodes are traditionally called "routers", when in fact they would be more appropriately called "forwarders".

Our concern is with the design of the forwarding algorithm. Clearly, the forwarding algorithm plays a critical role in the overall efficiency of the network and, therefore, it must be fast. In the context of a content-based network, a forwarding table contains *predicates* representing the conditions under which a message should be forwarded to a particular neighbor. The predicates in a router's forwarding table are formed from the primitive filters presented to the network by receivers associated with nodes downstream from that router in the message flow. Forwarding amounts to evaluating the predicates against the content of each message arriving at the router. We model message content as a set of attribute/value pairs, and a filter as a conjunction of constraints over the values of individual attributes. A predicate is then a disjunction of filters.

The algorithm we seek must be well behaved in situations where there are large numbers of filters and high volumes of messages. Notice that this is the inverse of the traditional database or document-retrieval problem, where solutions are driven by the need to match a single predicate (the query) against a large volume of data (the database). Rather than building indexes of the data, what we need to do here is find ways of indexing the predicates.

In this paper we present a forwarding algorithm and give the results of studies evaluating its performance. Because our goal is *fast* forwarding, our primary metric for success is how well we minimize, for a given message, the time it takes to identify the set of neighbors to which the message should be forwarded. Intuitively, the main scale factor of the algorithm is the total number of constraints resident in the forwarding table.

Our evaluation shows that the algorithm has good absolute performance under heavy loads and in a variety of network configurations, including the extreme case of a single, centralized router. It also shows that the algorithm scales sublinearly in the number of filters, with almost no degradation of throughput in the context of a network of routers with a fixed number of neighbor nodes. For example, on a 950Mhz processor, the algorithm is able to forward a 10-attribute message in 3 milliseconds in a situation where there are 20 predicates (i.e., neighbors) consisting of 250000 filters formed from 5 million individual constraints over an alphabet of 1000 attributes. In this experiment, the message went to 18 of the 20 neighbors, but we observed in other experiments that the performance generally improves (i.e., the forwarding time goes down) as the percentage of matching neighbors goes down. In terms of space, the forwarding table in this experiment occupies only 48 bytes per constraint, even though we have not yet turned our attention to optimizing that aspect of the algorithm.

In the next section we provide some necessary detail concerning content-based networking. We then discuss related work. Following that, we present our forwarding algorithm. An experimental evaluation of its performance is then described. We conclude with a summary and future plans.

2 Content-Based Networking

A content-based network is similar to a traditional address-based network in that it consists of *hosts* and *routers* connected by communication *links*. Hosts are nodes that have exactly one link, and act as senders or receivers of messages. Routers are nodes with more than one link, and act as dispatchers for messages that transit through them. Subsets of nodes may be directly connected to each other in *subnetworks*, thereby forming complete subgraphs, that are in turn connected to each other via routers. For simplicity, we ignore the internals of subnetworks and model them as single nodes. A schematic architecture of forwarding in

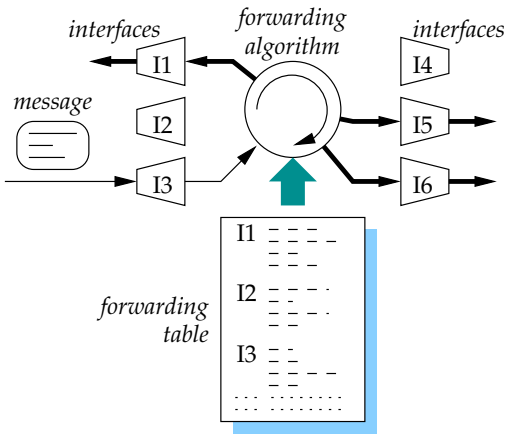


Figure 1: Forwarding in a Content-Based Network's Router

a content-based network's router is depicted in Figure 1. Following traditional networking terminology, we say that a router communicates with each neighbor node through an *interface*. The forwarding function processes an incoming message, consults the forwarding table, and determines the set of interfaces on which to output the message.

As mentioned in the previous section, it is the mode of communication in content-based networking that differs significantly from traditional (unicast or multicast) address-based networking. A content-based network is a network in which nodes are not assigned unique network addresses, nor are messages addressed to any specific node. Instead, each node advertises a *predicate* that defines messages of interest for that node and, thus, the messages that the node intends to receive. Again, the predicate advertised by a router represents the combined interests of the nodes downstream.

The concept of content-based network is independent of the form of messages and predicates. Denoting the universe of messages as $\mathcal{M}$, and the universe of predicates over $\mathcal{M}$ as $\mathcal{P} : \mathcal{M} \rightarrow \{true, false\}$, we say that $\mathcal{P}$ and $\mathcal{M}$ define a *content-based addressing scheme*, which in turn defines the content-based network. Consistently we say that the predicate p_n advertised by n is the *content-based address* of the node n. We also say that a message m is implicitly addressed by its content to a node n with content-based address p_n if $p_n(m) = true$.

In practice, we must refine these definitions somewhat. Here we use the concrete syntax and semantics embodied in the Siena event notification service [5, 6] to illustrate what we mean by messages and predicates. Note that in this regard, Siena is largely consistent with other publish/subscribe services [2, 3, 8, 9, 12, 15, 17]. Thus, a *message* is a set of typed attributes. Each attribute is uniquely identified within the message by a *name*, and has a *type* and a *value*. For purposes of this paper, we consider the common types *string*, *integer*, and *boolean*. For example, [*string* carrier = UA; *string* dest = MXP; *int* price = 600; *bool* upgradeable = true;] would be a valid message. A *predicate* is a disjunction of *filters*. A filter is a conjunction of *constraints* on individual attributes. Each constraint has a *name*, a *type*, an *operator*, and a *value*. A constraint defines an elementary condition over a message. A message matches a constraint if it contains an attribute with the same name and type, and if the value matches the condition defined by the operator and value of the constraint. For example, [*string* dest = MXP $\wedge$ *int* price $<$ 500] is a valid filter matching the message of the previous example.

3 Related Work and Contributions of this Paper

Yan and Garcia-Molina first suggested the idea of building indexes of predicates [16]. However, their context was quite different from ours. They were interested in better solutions to the SDI (selective dissemination of information) problem, which they characterize as finding the set of users interested in a document newly added to some collection of documents. An example application is a notification service for a digital library. The interests of users are represented by what are called *profiles*. The profiles are compared against a new document to determine a match. A document in this setting is treated as a set of words, and a profile is simply a conjunction of words. A profile matches a document if the document contains all the words in the profile. Yan and Garcia-Molina recognized that indexing the profiles (i.e., the predicates) could greatly speed up the matching function in the presence of large numbers of profiles. They developed a system called SIFT to demonstrate their ideas [17].

More recently, several researchers have studied the problem of evaluating a possibly large number of predicates against message-like data (as opposed to a large document) in the domain of filter matching for publish/subscribe systems. For this problem, various forms of decision trees and indexing structures for subscriptions have been proposed. These efforts can be classified into two broad categories based on the strategy used to search the predicate space.

The first approach is to start from the attribute constraints derived from the full set of subscriptions, and to move through them consulting the attributes appearing in the message. This approach is used in the form of a *matching tree* by Gough and Smith [11] and by Aguilera et al. [1]. It is also used in the form of a *binary decision diagram* by Campailla et al. [4].

The opposite approach is to start from the attributes of the message, and to move through them consulting the constraints. This is the approach used by Yan and Garcia-Molina in SIFT, if we consider a new document to be a "message" whose "attributes" are formed from the set of words appearing in the document. It is also the approach used by Fabret et al. in their publish/subscribe system Le Subscribe [9]. Le Subscribe goes beyond the SIFT indexing scheme by providing a main-memory matching algorithm that is "processor cache conscious", and by providing heuristic optimizations based on a clustering of subscriptions that share the same equality constraints over the same attributes.

Intuitively, if the first approach is used to implement a forwarding function, then the rough complexity of processing a single message is bound to the total number of constraints present in the forwarding table. Conversely, under the second approach, the rough complexity is bound to the number of attributes appearing in the message. In fact, based on published results, the second approach appears to perform better than the first in the presence of large numbers of subscriptions [9].

The work presented in this paper generally conforms to the second approach. We use the indexing data structure developed by Yan and Garcia-Molina as a starting point. In particular, we adopt their scheme for maintaining a global index of attribute constraints whose selection, based on the attributes of the input message, leads to the rapid identification of matching conjunctions. (Details are given in Section 4.) However, we have extended their ideas significantly, both to enhance the functionality of the matching algorithm and to make it appropriate for use in the forwarding function of a content-based network. The extensions include the following contributions.

- We extended the set of types and operators that can be used in predicates. SIFT is limited to strings (i.e., "words") and the equality operator over strings. Le Subscribe added integers and their associated relational operators. To this we have added the prefix, suffix, and substring operators for strings. These additional operators require the careful design and integration of whole new indexing structures.

- We added the explicit expression of disjunctions to the model of predicates. Both SIFT and Le Subscribe stop at the level of conjunctions of constraints. Disjunctions are critical for our purpose, since an interface (to use the terminology of Section 2) in a content-based network is inherently a disjunction.

- Given the presence of disjunctions, we developed a powerful optimization that can significantly shorten the processing steps needed to determine the set of outbound interfaces matching a given message and, thus, the neighbors to which the message should be forwarded. The optimization is based on the construction of what we call a *selectivity table*. The table is used to summarize for each

attribute in the alphabet of attributes, the subset of predicates for which those attributes are required for matching. For instance, in the predicate $(c_r \wedge c_s) \vee (c'_r \wedge c_t)$, where c_r and c'_r represent constraints on attribute r, and c_s and c_t represent constraints on attributes s and t, if the incoming message does not contain an attribute r, then we know immediately, without further processing, that the predicate cannot be matched. To give a flavor of the effectiveness of this optimization, our results for one set of experiments showed that for an alphabet of 1000 attributes used in 200000 predicates, we can eliminate 131000 of those predicates by examining just the first 10 of the 1000 entries in the table.

We hypothesize that the reason SIFT and Le Subscribe do not support explicit disjunctions (and, thus, the optimization of disjunctions) is that these systems were designed as centralized services. The consequence of this centralized view is that certain optimizations appropriate in a networking context, particularly those targeted at disjunctions, simply have no place in their design.

We note that Yan and Garcia-Molina describe an early prototype of a distributed servce [17]. However, the purpose of that service is to offer increased reliability through replication, not act as a network of store-and-forward routers. Further, we note that our ideas are largely complementary to those of Le Subscribe, and we plan in the future to look at adopting what we can of their ideas into our implementation.

We close this section with a remark about experimentation in this domain. The existing algorithms and index structures that have been developed for predicate matching are quite complex. Unfortunately, only incomplete descriptions of them have been provided in the literature. Moreover, we were not able to obtain the implementations that were used to produce the reported results. This makes any kind of rigorous comparative analysis virtually impossible. On the other hand, the significance of such an analysis would be questionable anyway, since the domain of forwarding in a content-based network is not the one for which these other systems, such as SIFT and Le Subscribe, were designed. For these reasons, we restrict the evaluation presented in Section 5 to experiments on the viability of the specific data structures and optimizations that we have developed, and use absolute performance as the universal metric by which our results should be judged. In addition, we have made both the implementation and the experimental data publicly available[1] so that others hoping to improve on our ideas have a concrete basis upon which to conduct meaningful experiments.

4 Forwarding Algorithm

Recalling the definitions of Section 2, a forwarding table is a one-to-one association of *predicates* to *interfaces*. A predicate is a disjunction of *filters*, a filter is a conjunction of elementary *constraints*, and a constraint is a quadruple $\langle$*type*, *name*, *op*, *value*$\rangle$. An example of the logical content of a forwarding table is shown in Figure 2a, where I_s is an interface and $\mathrm{f}_{s.t}$ is a filter in the disjunction of filters associated with interface I_s. Constraints on individual attributes within a filter are shown in the third column of the table. Although not evident in this particular example, identical filters can in general be associated with more than one interface, just as identical constraints can be associated with more than one filter.

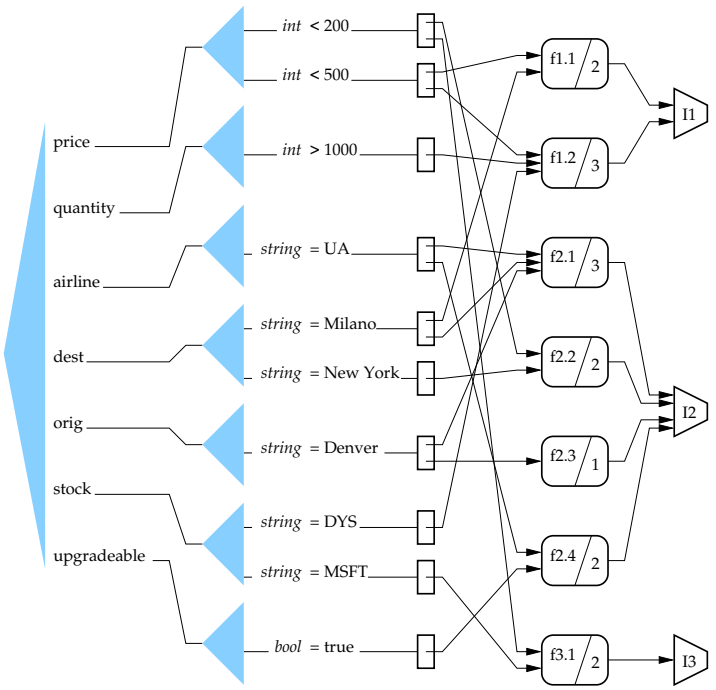
Forwarding an incoming message m amounts to computing the set of interfaces associated with a predicate matching m. Because each interface is associated with exactly one predicate, in the following discussion we use the terms interface and predicate interchangeably. The forwarding function that we have developed is an evolution of a basic matching algorithm, which in turn is founded on a particular index structure representing the forwarding table. This basic algorithm is known as the *counting algorithm*, and has been applied to the matching problem in the context of centralized publish/subscribe systems [9, 16]. As mentioned in Section 3, these previous applications of the counting algorithm have only dealt with predicates that were conjunctions, not disjunctions.

In order to describe our extended version of the algorithm, we first give a high-level view of the structure of the forwarding table and then introduce a simple variant of the counting algorithm that handles disjunctions. This variant can already exploit the presence of disjunctions to optimize the matching process somewhat. Further optimizations are described in sections 4.2 and 4.3.

[1] `http://www.cs.colorado.edu/serl/siena/`

I_1	$f_{1.1}$	*string dest = Milano* *int price < 500*
	$f_{1.2}$	*string stock = DYS* *int quantity > 1000* *int price < 500*
I_2	$f_{2.1}$	*string airline = UA* *string orig = Denver* *string dest = Milano*
	$f_{2.2}$	*string dest = New York* *int price < 200*
	$f_{2.3}$	*string orig = Denver*
	$f_{2.4}$	*string airline = UA* *bool upgradeable = true*
I_3	$f_{3.1}$	*string stock = MSFT* *int price < 200*

(a)



(b)

Figure 2: Example Contents of a Forwarding Table and its Representation

4.1 Forwarding Table and the Extended Counting Algorithm

At a high level, the forwarding table is organized as a two-part, left-to-right structure. The left-hand side is an index of all the individual constraints found in all the predicates associated with all the neighbors of the content-based router. The outputs of the left-hand side (i.e., the individual constraints lying at the ends of index paths) are connected as boolean inputs to the right-hand side of the structure; if the algorithm arrives at a particular constraint after traversing the index, then the constraint has been found to be true for the message. The right-hand side implements a network of logical connections representing the conjunctions of constraints into filters and the disjunction of filters into interfaces. Figure 2b shows a schematic view of that data structure for the example of Figure 2a.

Based on this general data structure, the counting algorithm proceeds as follows. For a given message m, it iterates through the attributes $a_1, a_2, \ldots, a_k$ of m. For each attribute a_i, it finds the constraints $c_{i,1}, c_{i,2}, \ldots c_{i,n_i}$ matched by a_i using the left-hand-side index of the forwarding table. Then, iterating through all the matched constraints $c_{1,1}, c_{1,2}, \ldots c_{1,n_1} \ldots c_{k,1}, c_{k,2}, \ldots c_{k,n_k}$, it finds the matched filters using the right-hand-side boolean network.

Figure 3 shows the counting algorithm in pseudocode. The algorithm uses two running data structures to maintain state during the matching process. The first structure is a table of counters (hence the name of the algorithm) for partially-matched filters. The second data structure is a set containing the interfaces to which the message should be forwarded. For each constraint found through the constraint index, the algorithm increments the counter of all the filters linked from that constraint. When a counter associated with filter f reaches the total number of constraints linked to f, the filter is satisfied and the algorithm adds the interface linked from f to the set of matched interfaces.

The main difference between this extended counting algorithm and its more basic counterpart is that having disjunctions of filters allows us to use the set of matched interfaces to shortcut the evaluation of the filters. In particular, we can eliminate a lookup in the table of counters for all the filters linked to an interface that has already been matched. Moreover, as an additional shortcut, we can terminate the execution of the whole process for a message as soon as the set of matched interfaces contains the complete set of neighbor interfaces, since we know that further processing cannot provide any additional information. Notice that

```
proc counting_forward(message m) {
  map<filter,int> counters = ∅
  set<interface> matched = ∅
  foreach a in m {
    set<constraint> C = matching_constraints(a)
    foreach c in C {
      foreach f in c.filters {
        if f.interface ∈ matched {
          if f ∉ counters {
            counters := counters ∪ ⟨f,0⟩
          }
          counters[f] := counters[f] + 1
          if counters[f] = f.size {
            output(m,f.interface)
            matched := matched ∪ {f.interface}
            if matched = total_interface_count {
              return } } } } } }
}
```

Figure 3: Pseudocode of the Counting Algorithm

all the sets and set operations in the forwarding algorithm can be implemented very efficiently with bit vectors.

Our extended counting algorithm is really only the starting point for the optimizations leading to a fast forwarding algorithm. In the remainder of this section we describe two of those optimizations, each focussing on one of the two sides of the forwarding table. Evaluations of the effectiveness of the optimizations are provided in Section 5.

4.2 Multi-Operator Index

The index forming the left-hand side of the forwarding table is meant to speed up the process of finding the constraints that are satisfied by an attribute of an incoming message. Obviously, the first stage in the index should be based on the constrained attribute's name and type. In our implementation we use a straightforward combination of a standard ternary search trie (TST) for the strings representing attribute names, and then a simple switch on the type.

Once we have selected the right name and type, we use subindexes that exploit specific properties of each constraint operator. For example, indexing equality constraints is an immediate application of traditional indexes, since it is equivalent to indexing values. To index less-than constraints on integer attributes $\langle int, \text{name}, <, k_1\rangle$, $\langle int, \text{name}, <, k_2\rangle$, $\ldots \langle int, \text{name}, <, k_n\rangle$, we can simply maintain the constraints ordered according to their constant value (a sorted vector would suffice). Then, to find all the constraints matched by a value of x, we simply walk through the index going from the bottom to the top, finding all the constraints with constant value greater than x. A similar structure can be used for greater-than constraints. Notice that this implementation of indexes for integer constraints is optimal, since the complexity of the search operation is on the order of the size of the output (i.e., the number of matching constraints) plus the complexity of an exact match search in a table that can be optimized using well-known searching techniques. Figure 4 depicts an index for integer constraints processed against a corresponding attribute in a message.

As mentioned in Section 3, previous algorithms were designed without consideration of the prefix, suffix, and substring operators for strings. We could use the same approach for these operators as we do for the integer operators, namely to create and maintain a separate subindex for each. In fact, this was what we did in an earlier version of our algorithm. However, that approach is not optimal for strings, since it requires a potentially large number of comparisons. In order to support the fuller set of string operators

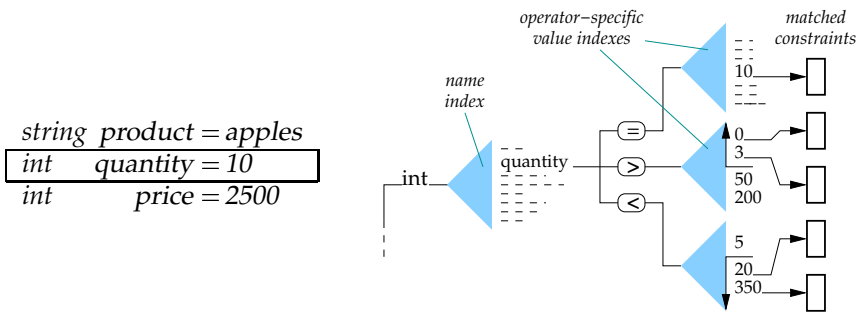


Figure 4: Example of Integer Constraint Indexing

with a very efficient index structure, we developed a *multi-operator index* for string constraints capable of supporting equality, less-than, and greater-than operators, as well as the prefix, suffix, and substring operators all in a single index. The basic skeleton of this index is a TST that we have extended in three ways.

- We added the capability of matching partial strings, which we use to represent prefix and substring constraints.
- We added a pair of *crown* lists, which we use to link the sequence of less-than and greater-than constraints inserted as leaves in the TST.
- We added a pair of backtrack functions that are necessary to move from a partial match to the (alphabetically) closest complete match, which we use to jump to the less-than and greater-than chains.

An example of this extended TST is depicted in Figure 5. In addition to the basic TST structure, we have two types of nodes, representing partial and complete matches, respectively. Nodes that represent partial matches link prefix (*pf*) and substring (*ss*) constraints. Nodes that represent complete matches link equals (=), less-than (<), greater-than (>), and suffix (*sf*) operators. Nodes representing complete matches are also cross linked through two singly-linked lists representing the chains of less-than and greater-than operators. They link every complete match node to the nearest less-than or greater-than constraint. The less-than chain is depicted in Figure 5 using a dashed line. For sake of clarity, only the less-than chain is shown.

The lookup function starts from the first character of the input string, and uses a slightly modified TST lookup subfunction. This subfunction recognizes partial matches along the path through the TST, in addition to final, complete matches. When a partial-match node is reached, the function returns the prefix constraint and/or the substring constraint associated with that partial match. The function also returns a pointer to the corresponding internal node and a pointer to the position reached within the input string. These two pointers can then be passed to the same lookup function to continue the search from the previous partial match node. At some point, this process will terminate, either because it cannot move forward in the TST and/or because it has reached the end of the input string. If the final node touched by the lookup process is a leaf node of the TST (i.e., if it contains the terminator character #), then the lookup function returns the corresponding equality, inequality, and suffix constraints. If the terminal node is not a leaf (i.e., if no complete match was found), then the lookup function backtracks to the two closest leaf nodes, one preceding and the other following the final node in alphabetical order. From a matching final node or from the closest matching nodes, the lookup function can immediately jump onto the less-than and greater-than chains, reaching one-by-one all the matching less-than and greater-than constraints.

The lookup process is then repeated for each character of the input string, ignoring prefix, equality, less-than, and greater-than operators. This iteration allows us to identify all the substring and suffix constraints.

The complexity of the complete lookup function is on the order of l times the complexity of a TST search plus the size of the output—that is, $O(l(logN + l) + -result\tilde{}$, where l is the length of the input string, and N is the number of strings in the TST.

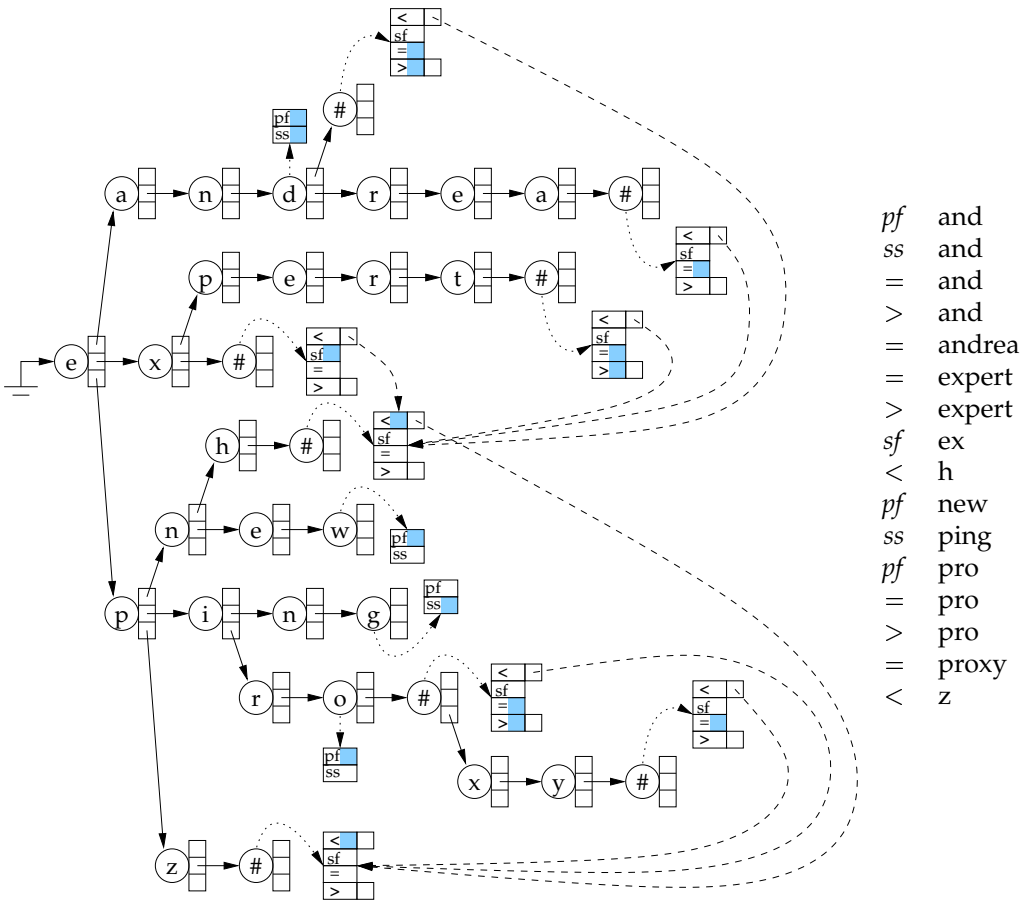


Figure 5: Extended TST that Implements a Multi-Operator Index for String Constraints

4.3 Exploiting Attribute Selectivity

Intuitively, we can save time in processing a message if we can eliminate interfaces from consideration as soon as possible. Eliminating an interface can mean eliminating the evaluation of potentially many filters, and in turn, potentially many constraints. We can see that our counting algorithm already succeeds in doing this to a certain degree, by making use of the set of matched interfaces, as described in Section 4.1.

However, we can go futher than this, based on the following reasoning. Let us call an attribute name a the *determinant* attribute name for interface I, if every filter of I contains at least one constraint on a. Because filters are conjunctions, if a message does not contain an attribute a, then the message cannot possibly be forwarded through interface I and so can be ignored during the processing of the message. For example, in the forwarding table of Figure 2, "price" is a determinant for interfaces I_1 and I_3, "stock" is a determinant for I_3, and no other attribute name is a determinant for any other interface.

We use the concept of a determinant to pre-compute from the set of predicates what we call a *selectivity table*, which is a map that associates attribute names with the interfaces for which that attribute name is a determinant. Computing the selectivity table is straightforward, and amounts to computing the intersection of the attribute names of all the filters for each interface. Note that it would also be straightforward to combine the name with a type for the purpose of establishing selectivity, but to keep the presentation simple, we do not discuss this point.

When a message arrives, we can use the table to easily calculate—without incurring the cost of traversing the forwarding table—an initial set of interfaces that will not match the message. We perform this pre-processing step by iterating through the selectivity table, excluding all the interfaces associated with determinant attributes that are not present in the message. In the subsequent processing of the message, we use this set exactly in the same way we use the set of matched interfaces. In fact, we use a single set to annotate both excluded and already matched interfaces. The resulting algorithm is identical to the counting algorithm, except for the introduction of the selectivity processing step (Figure 6).

```
proc selectivity_forward(message m) {
  map<filter,int> counters = ∅
  set<interface> ifs_to_ignore = pre_process(m)
  foreach a in m {
    set<constraint> C = matching_constraints(a)
    foreach c in C {
      foreach f in c.filters {
        if f.interface ∈ ifs_to_ignore {
          if f ∉ counters
            counters := counters ∪ ⟨f,0⟩
          counters[f] := counters[f] + 1
          if counters[f] = f.size {
            output(m,f.interface)
            ifs_to_ignore := ifs_to_ignore ∪ {f.interface}
            if ifs_to_ignore = total_interface_count
              return } } } } }
}
```

```
map<name, set<interface>> selectivity
int pre_processing_rounds

proc pre_process(message m) {
  set<interface> result = ∅
  int rounds = pre_processing_rounds
  foreach ⟨a,s⟩ in selectivity {
    if rounds = 0
      return result
    rounds := rounds − 1
    if a ∉ m {
      result := result ∪ s
      if result = total_interface_count
        return result
    } }
  return result
}
```

Figure 6: Pseudocode of the Forwarding Algorithm with Selectivity Table and Pre-Processing

In order to maximize the effectiveness of the pre-processing, we sort the selectivity table in descending order by the cardinality of the set of excluded interfaces. This way the pre-processing function will always encounter the most selective names first. We parameterize the pre-processing function by the number of pre-processing *rounds*, by which we mean how far down the selectivity table the pre-processing function will traverse. With zero rounds, the selective forwarding algorithm reduces to the counting algorithm, since the selectivity table is ignored. As we increase the number of rounds we should see an increase in the number of interfaces determined to be safely ignored, which should in turn lead to a performance improvement. However, adding more and more pre-processing rounds, and thus going farther and farther down in the

selectivity table, will add fewer and fewer additional interfaces, at the cost of more and more lookup operations on the message and set-union operations on the data structure. Therefore, we would expect to see a continuous decline in performance beyond a certain point. This is borne out by the experiments described in the next section.

5 Evaluation

In order to evaluate our algorithm, we implemented it and studied its performance using a series of synthetic workloads derived from various combinations of filters, interfaces, and messages. In this section we present the results of our evaluation. As mentioned in Section 3, we have made our implementation, as well as our workload generator, available on line for others to use.

5.1 Experimental Setup and Parameters

We implemented our algorithm in C++ and ran all the experiments on a 950Mhz computer with 512Mb of main memory. In addition to the main algorithm and data structures, we created some auxiliary programs to generate parameterized loads of filters and messages. In particular, we have identified and used the parameters listed in Table 1. We performed all the experiments with 100 messages ($M = 100$), each one having between 1 and 19 attributes ($a_l = 1$,$a_h = 19$, and an average $\overline{a} = 10$).

M	number of messages
a_l, a_h	number of attributes per message, uniform in $[a_l, a_h)$ range
I	number of interfaces (= number of predicates)
f_l, f_h	number of filters per interface, uniform in $[f_l, f_h)$ range
c_l, c_h	number of constraints per filter, uniform in $[c_l, c_h)$ range
D_a	distribution function for attribute names
D_c	distribution function for constraint names
D_t	distribution function for data types in both filters and messages
D_{os}	distribution function for operators in string constraints
D_{oi}	distribution function for operators in integer constraints
D_s	distribution function for string values in both filters and messages
D_i	distribution function for integer values in both filters and messages

Table 1: Scenario Definition Parameters

Roughly speaking, the primary measure of scalability is the "size" of the forwarding table, which is well characterized by the total number of elementary constraints $C \approx I \times \overline{f} \times \overline{c}$, where $\overline{f} = \frac{1}{2}(f_l + f_h - 1)$ and $\overline{c} = \frac{1}{2}(c_l + c_h - 1)$. We experimented with forwarding tables of up to 5 million constraints, distributed in various ways among filters, and filters among interfaces. Specifically, we fixed the range of constraints per filter, with $c_l = 1$ and $c_h = 10$, and we used different numbers of interfaces I and different ratios of filters per interface, maintaining $f_l = 1$ and varying f_h. The choice of a fixed range of constraints per filter, with an average of 5 constraints per filter, is based on practical considerations on the type of filters we expect to be posed by typical end users. The number of interfaces I gives an indication of the characteristics of a router, its position, and it role in the larger content-based network. The resulting total number of filters $F = I \times \frac{1}{2} f_h$ is a rough measure of the total size of the network in terms of nodes and end users, and therefore an important measure of scalability.

For attribute names, we experimented with a set of 1000 elements ($D_a = 1000$). In order to use realistic names, we composed our sample sets by selecting random words out of a common dictionary, and we weighted our set of names using a Zipf distribution. We then used the same set of words for both attributes in messages and constraints in filters (and therefore $D_c = D_a$). Notice that while this may be a simplification in defining the experiments, in fact it produces the most challenging scenarios for a forwarding algorithm. The reason is that having two completely overlapping sets of names maximizes the chances of having matching attributes and constraints. In the opposite, extreme case of two completely disjoint name sets

(one for attributes, and one for constraints) there would be no matches at all, and the time complexity for the forwarding algorithm would be $O(\overline{a} \log C)$.

For attribute values, we used a combination of dictionary values for strings and a range for integers. For strings, we compiled a list of words by extracting 1000 words from the dictionary. For integer values we used a range of 100 values. For both integers and strings, we used a uniform distribution to select values. We used the same distribution of string and integer values for both the values in messages and the values in constraints. Notice once again that having a unified set of values increases the possibilities of having positive matches between constraints and attributes, thereby adding complexity to the matching process.

For attribute types and constraint types we used the same distribution D_t: 50% strings and 50% integers. For operators in integer constraints, the distribution D_{oi} was 60% equality, 20% less-than, and 20% greater-than. Finally, for operators in string constraints, the distribution D_{os} was 35% equality, 15% prefix, 15% suffix, 15% substring, 10% less-than, and 10% greater-than.

5.2 Basic Results

Figure 7 shows a summary of the results of our experiments. The graphs show the matching time per message over the total number of constraints in the forwarding table, which ranges from a few hundred to over five million. Every graph shows pairs of curves, representing the cases with no selectivity table pre-processing rounds ($r = 0$) and with 10 rounds ($r = 10$), respectively.

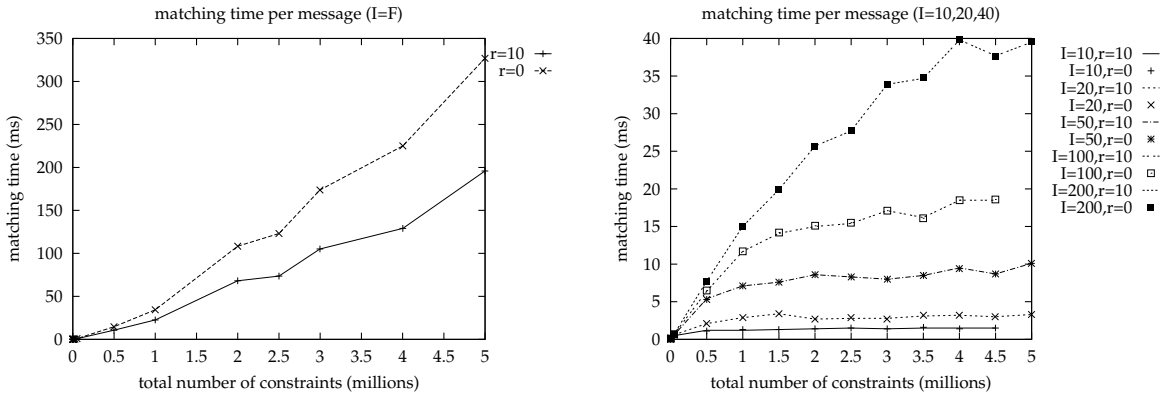


Figure 7: Performance of the Forwarding Algorithm. Centralized Architecture (a) and Distributed Architectures (b)

Figure 7a represents the degenerate situation in which we model a centralized server, namely where the forwarding table has exactly one filter per interface. This is the worst case for our algorithm, yet its performance is arguably quite reasonable, taking only about 330 milliseconds to match a message in the presence of over 5 million constraints, corresponding to more or less 1 million filters and 1 million interfaces. The most important observation we make concerning the graph of Figure 7a is that our optimization based on the selectivity table is particularly effective in this extremely difficult case, achieving a reduction of matching time of up to 40%.

Figure 7b shows that the performance of the forwarding algorithm in scenarios more closely modeling a network of content-based routers, with a fixed number of interfaces, is significantly better, both in the absolute values and in the general sublinear behavior. Notice that in these cases, the curves with zero rounds are essentially indistinguishable from the corresponding ones with 10 rounds of pre-processing. On the one hand, this says that our optimization has no effect in configurations with a high ratio of filters per interface. This is not surprising, since in the presence of many filters for each interface, it becomes highly unlikely for an attribute name to be present in all the filters of an interface, thus reducing the overall selectivity of each individual attribute. On the other hand, from the same observation, we can conclude that

our optimization does not add measurable cost, even in cases in which the simple version of the algorithm is already extremely fast.

It would seem that a fundamental parameter in determining the behavior of the algorithm would be the ratio of filters to interfaces, given a fixed total number of constraints (i.e., essentially a fixed forwarding table size). This intuition is confirmed by the results shown in Figure 8. The graph expresses very well the effect of handling disjunctions, and in particular, it shows that the forwarding algorithm performs at its best with very large disjunctions. On the left-hand side of Figure 8, we see once again the positive effect of the exclusion of interfaces using the selectivity table.

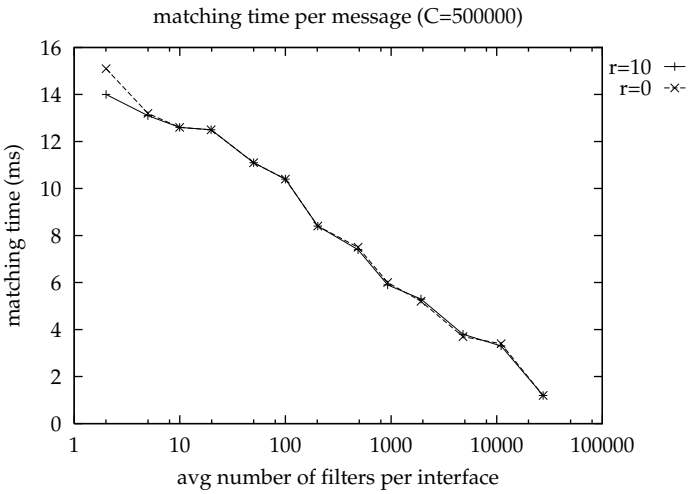


Figure 8: Performance of the Forwarding Algorithm with a Varying Ratio of Filters per Interface

5.3 Sensitivity to the Number of Pre-Processing Rounds

The idea of using the selectivity table is to reduce the processing time by pre-selecting interfaces that can be safely ignored during the forwarding function. The exact amount of this reduction depends essentially on the combination of two independent factors. The first is the level of selectivity of each name, which is purely a characteristic of the predicate set. The second is the number of pre-processing rounds. As we point out in Section 4.3, the initial effect of adding rounds is to exclude more interfaces from the main processing function. However, after a certain point, this effect should fade due to the reduced selectivity of names farther down in the selectivity table, and because each round adds a certain processing cost that depends on the number of interfaces.

The tension between cost and benefit of the pre-processing function is exemplified by the experimental results shown in Figure 9. The two curves represent the advantage of the pre-processing function as a (percentage) performance gain, over the simple counting algorithm. The curve that shows the highest advantage corresponds to the case of 5 million constraints. All the experiments are run over forwarding tables with one filter per interface. The experiments show that the pre-processing function becomes ineffective and ultimately a cost after 50 to 70 rounds. We performed all our experiments using 10 rounds.

5.4 Network Effect

The experiments discussed above evaluate the performance of our forwarding algorithm, but only by examining an individual router. The question that arises is whether a true network of routers would out perform a centralized server under the same heavy workload. We can answer this question by comparing the end-to-end latency induced by the forwarding function in two different scenarios: the first with a single server, and the second with a combination of interconnected routers. In both cases, we consider a total of

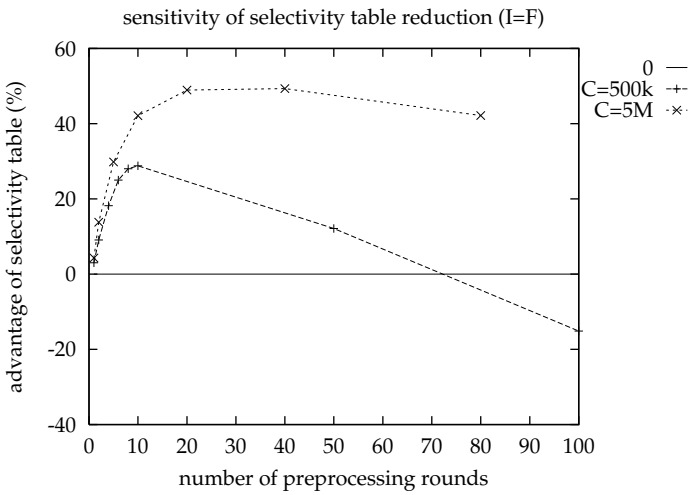


Figure 9: Cost/Benefit Analysis of the Pre-Processing Function with Varying Number of Rounds

1 million filters formed from 5 million constraints, where each filter is associated with a distinct destination. Notice that this configuration represents the worst case for our optimizations, so we would expect the performance to be better in practice.

The first configuration corresponds to the curve for $I = F$ from Figure 7. In this case, the latency is about 350 milliseconds, which corresponds to the matching time of one run of the forwarding function over the complete set of filters. The second configuration can be obtained by connecting the destination nodes through a set of routers with a limited number of interfaces. Using routers with I interfaces, interconnected in an appropriate configuration, we can reach H destinations with at most $2 \log_I H$ hops. In our example, using routers with 20 interfaces, we can span the network in at most 12 hops, which would give a worst-case total latency of only about 40 milliseconds, as shown by the curve for $I = 20$ in Figure 7b. This clearly demonstrates the viability of using a network of routers that uses an appropriately optimized forwarding function.

5.5 Summary of Evaluation

Our experiments have shown that our forwarding algorithm has good absolute performance and good cost amortization over a variety of loads. In particular, we found that:

- the basic *short-circuit evaluation* of filters greatly reduces processing time in the case where a single message may match a large number of filters;
- the use of the *selectivity table* improves the ability to short-circuit the forwarding function, reducing the matching time up to 40% in the critical cases of routers with numerous interfaces and especially in the extreme case of centralized routers; and
- the use of the *selectivity table* has no measurable costs over the basic algorithm.

In summary, the selectivity table proved to reduce forwarding costs in the most critical cases, without adding any penalties in other cases in which the simple matching algorithm already offers good performance. We conclude from the evaluations that our forwarding algorithm is viable under heavy loads, and that the optimizations we proposed have significant, positive effects.

6 Conclusion

In this paper we have presented the first algorithm designed specifically for the implementation of the forwarding function of routers in a content-based network. The algorithm is based on earlier work in the area of centralized content filtering of both large documents and small messages. Our algorithm refines, adapts, and extends this work for use in a very different context. We formulated a variant of the counting algorithm that can handle disjunctive predicates, and developed optimizations targeted specifically at the disjunctions that are the semantics of network interfaces in a content-based network.

In order to evaluate the algorithm, we created an implementation in C++ and subjected it to a battery of synthetic workloads. From these experiments we found that the algorithm has good overall performance. The experiments also confirmed the validity of our optimization techniques, and the stability of the algorithm even in circumstances that are suboptimal for the optimizations. We have made our implementation available on line for reference and further evaluation by other researchers.

In the immediate future we plan to integrate our algorithm into our prototype content-based network architecture. As a natural progression of this work, we plan to attack the hard problem of routing in a content-based network. Derived from our experience developing the Siena wide-area event notification service, we have already defined the basic concepts of content-based subnetting and supernetting, and we have implemented what amounts to a routing table. Using that as a basis, we plan to study and develop optimized data structures for routing, as well as efficient and robust routing protocols for content-based networks.

Acknowledgments

The authors would like to thank Jing Deng for his contributions to an earlier version of the forwarding algorithm, and to Matthew Rutherford for his help in implementing the algorithm. The work of the authors was supported in part by the Defense Advanced Research Projects Agency, Air Force Research Laboratory, Space and Naval Warfare System Center, and Army Research Office under agreement numbers F30602-01-1-0503, F30602-00-2-0608, N66001-00-1-8945, and DAAD19-01-1-0484. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Air Force Research Laboratory, Space and Naval Warfare System Center, Army Research Office, or the U.S. Government.

References

[1] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Eighteenth ACM Symposium on Principles of Distributed Computing (PODC '99)*, pages 53–61, Atlanta, Georgia, May 4–6 1999.

[2] G. Banavar, T. D. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *The 19th IEEE International Conference on Distributed Computing Systems (ICDCS '99)*, pages 262–272, Austin, Texas, May 1999.

[3] L. F. Cabrera, M. B. Jones, and M. Theimer. Herald: Achieving a global event notification service. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, Elmau, German, May 2001.

[4] A. Campailla, S. Chaki, E. Clarke, S. Jha, and H. Veith. Efficient filtering in publish-subscribe systems using binary decision diagrams. In *Proceedings of the 23th International Conference on Software Engineering*, pages 443–452, Toronto, Canada, May 2001.

[5] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proceedings of the Nineteenth ACM Symposium on Principles of Distributed Computing (PODC 2000)*, pages 219–227, Portland, Oregon, July 2000.

[6] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, Aug. 2001.

[7] A. Carzaniga and A. L. Wolf. Content-based networking: A new communication infrastructure. In *NSF Workshop on an Infrastructure for Mobile and Wireless Systems*, Scottsdale, Arizona, Oct. 2001. In conjunction with the International Conference on Computer Communications and Networks ICCCN.

[8] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transactions on Software Engineering*, 27(9):827–850, Sept. 2001.

[9] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *ACM SIGMOD 2001*, pages 115–126, Santa Barbara, California, May 2001.

[10] M. Gitter and D. R. Cheriton. An architecture for content routing support in the Internet. In *3rd USENIX Symposium on Internet Technologies and Systems*, pages 37–48, San Francisco, California, Mar. 2001.

[11] J. Gough and G. Smith. Efficient recognition of events in a distributed system. In *Proceedings of the 18th Australasian Computer Science Conference*, Adelaide, Australia, Feb. 1995.

[12] Object Management Group, Inc., Needham, Massachusetts. *Notification Service*, Aug. 1999.

[13] D. S. Rosenblum and A. L. Wolf. A design framework for Internet-scale event observation and notification. In *Proceedings of the Sixth European Software Engineering Conference*, number 1301 in Lecture Notes in Computer Science, pages 344–360. Springer–Verlag, 1997.

[14] B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings of AUUG97*, pages 243–255, Brisbane, Australia, Sept. 3–5 1997.

[15] Sun Microsystems, Inc., Mountain View, California. *Java Message Service*, Nov. 1999.

[16] T. W. Yan and H. Garcia-Molina. Index structures for selective dissemination of information under the Boolean model. *ACM Transactions on Database Systems*, 19(2):332–364, June 1994.

[17] T. W. Yan and H. Garcia-Molina. The SIFT information dissemination system. *ACM Transactions on Database Systems*, 24(4):529–565, Dec. 1999.