An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems

Guruduth Banavar, Tushar Chandra, Bodhi Mukherjee, Jay Nagarajarao,
Robert E. Strom, and Daniel C. Sturman
IBM T. J. Watson Research Center, 30 Saw Mill River Road, Hawthorne, NY 10532
{banavar, tushar, bodhi, jayrao, strom, sturman}@watson.ibm.com

Abstract

The publish/subscribe (or pub/sub) paradigm is an increasingly popular model for interconnecting applications in a distributed environment. Many existing pub/sub systems are based on pre-defined subjects, and hence are able to exploit multicast technologies to provide scalability and availability. An emerging alternative to subject-based systems, known as content-based systems, allow information consumers to request events based on the content of published events. This model is considerably more flexible than subject-based pub/sub. However, it was previously not known how to efficiently multicast published events to interested content-based subscribers within a large and geographically distributed network of broker (or router) machines. In this paper, we develop and evaluate a novel and efficient distributed algorithm for this purpose, called "link matching". Link matching performs just enough computation at each node to determine the subset of links to which an event should be forwarded. We show via simulations that (a) link matching yields higher throughput than flooding when subscriptions are selective, and (b) the overall CPU utilization of link matching is comparable to that of centralized matching.

1 Introduction

The *publish/subscribe* paradigm is an easy to use and efficient paradigm for interconnecting applications in a distributed environment. Pub/sub based middleware is currently being applied for application integration in many domains including financial, process automation, transportation, and mergers and acquisitions. Pub/sub systems contain information providers, who publish events to the system, and information consumers, who subscribe to particular categories of events within the system. The system ensures the timely delivery of published events to all interested subscribers. A pub/sub system is implemented on a network of *event brokers* that are responsible for routing events between publishers and subscribers.

The earliest pub/sub systems were *subject-based*. In these systems, each unit of information (which we will call an *event*) is classified as belonging to one of a fixed set of *subjects* (also known as groups, channels, or topics). Publishers label each event with a subject; consumers subscribe to all the events within a particular subject. For example a subject-based pub/sub system for stock trading may define a group for each stock issue; publishers may post information to the appropriate group, and subscribers may subscribe to information regarding any issue. In the past decade, systems supporting this paradigm have matured significantly resulting in several academic and industrial strength solutions [4][10][12][13][15]. A similar approach has been adopted by the OMG for CORBA event channels [11].

An emerging alternative to subject-based systems is content-based subscription systems [14]. These systems support a number of *information spaces*, each associated with an *event schema* defining the type of information contained in each event. Our stock trade example (shown in Figure 1) may be defined as a single information space with an event schema defined as the tuple [issue: string, price: dollar, volume: integer]. A content-based subscription is a predicate against the event schema of an information space, such as (issue="IBM" & price < 120 & volume > 1000) in our example.

With content-based pub/sub, subscribers have the added flexibility of choosing filtering criteria along multiple dimensions, without requiring pre-definition of subjects. In our stock trading example, the subject-based subscriber is forced to select trades by issue name. In contrast, the content-based subscriber is free to use an orthogonal criterion, such as volume, or indeed a collection of criteria, such as issue, price and volume. Furthermore, content-based pub/sub reduces the administrative overhead of defining and maintaining a large number of groups, thereby making the system easier to manage. Finally, content-based pub/sub is more general in that it can be used to implement subject-based pub/sub, while the reverse is not true. While content-based pub/sub

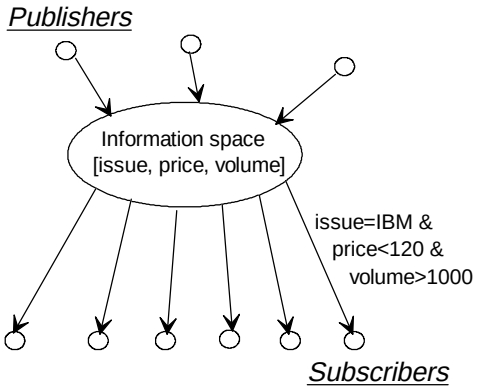


Figure 1: Pub/Sub information space

is the more powerful paradigm, efficient and scalable implementations of such systems have previously not been developed.

In order to efficiently implement a content-based pub/sub system, two key problems must be solved:

- The problem of efficiently *matching* an event against a large number of subscribers on a single event broker.
- The problem of efficiently *multicasting* events within a network of event brokers. This problem becomes crucial in two settings: 1) when the pub/sub system is geographically distributed and event brokers are connected via a relatively low speed WAN (compared to high-speed LANs), and 2) when the pub/sub system has to scale to support a large number of publishers, subscribers and events. In both cases it becomes crucial to limit the distribution of a published event to only those brokers that have subscribers interested in that event.

One of the strengths of subject-based pub/sub systems is that both problems are amenable to straightforward solutions: the matching problem is solved using a simple table lookup; the multicast problem is solved by defining a multicast group per subject, and multicasting each event to the appropriate multicast group. For content-based pub/sub systems, however, previous literature does not contain solutions to either problem. In this paper we present the first efficient solution to the multicast problem for content-based pub/sub. In a companion paper [2] we present an efficient solution to the matching problem for these systems.

There are two straightforward approaches to solving the multicasting problem for content-based systems: (1) The *match-first* approach, where the event is first matched against all subscriptions, thus generating a destination list and the event is then routed to all entries on this list; and (2) The *flooding* approach, where the event is broadcast, or flooded, to all destinations using standard multicast technology, and unwanted events are then filtered out at these destinations. Both approaches may work well in small systems, but can be inefficient in a large system with thousands of potential destinations. In a match-first system, if after matching, the broker sends events directly to each destination via separate point-to-point connections, we may have multiple copies of the same event going over the same network link on its way to multiple remote subscribers. On the other hand, if the broker appends the destination list to the event header to enable intermediate routers to route efficiently, then the size of the event header may be impractically large. The flooding approach suffers when, in a large system, only a small percentage of clients want any single event and many brokers receive the event unnecessarily. Moreover, the flooding technique cannot exploit *locality* of information requests, i.e., when clients in a single geographic area are likely to have more similar requests for data than more remote clients.

The central contribution of this paper is a new distributed algorithm for *content-based routing*, an efficient solution to the multicast problem for content-based pub/sub systems. With this algorithm, called *link matching*, each broker performs just enough of the matching work to determine which neighboring brokers should receive the event, and then forwards the event along links to these neighbors. The disadvantages of the match-first approach are avoided since no additional information is appended to the event headers. Further, at most one copy of a event is sent on each link. The disadvantages of the flooding approach are avoided as the event is only sent to brokers and clients needing the event, thus exploiting locality. We illustrate, using a network simulator, that flooding overloads the network at significantly lower publish rates than link matching. We also describe our implementation of a distributed Java based prototype of content-based pub/sub brokers.

The remainder of this paper is organized as follows. In section 2, we summarize our solution to the matching problem (i.e., the case when the network consists of a single broker). In section 3, we discuss how to extend the solution to the matching problem into a link-matching solution to the content-based routing problem in a multi-broker network. In section 4, we evaluate the performance of this approach and compare it to the flooding approach. Sections 5 and 6 provide discussion of related work and point to avenues of future work respectively.

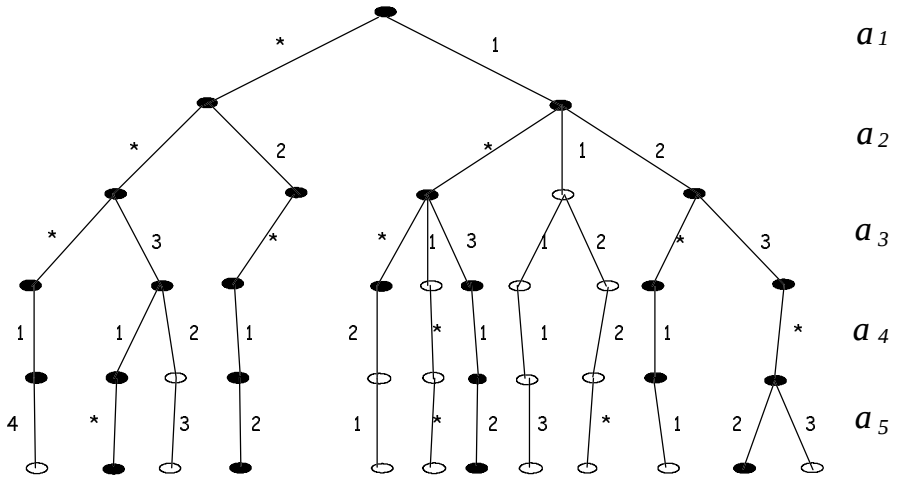


Figure 2: Parallel search tree for matching events to subscriptions

2 The Matching Algorithm

This section summarizes a non-distributed algorithm for matching an event against a set of subscriptions, and returning the subset of subscriptions that are satisfied by the event. (A more detailed presentation of matching along with experimental and analytic measures of performance are the subject of a companion paper [2].) This matching algorithm is the basis of our distributed multicast protocol, presented in the following section.

Our approach to matching is based on sorting and organizing the subscriptions into a parallel search tree (PST) data structure, in which each subscription corresponds to a path from the root to a leaf.[1] The matching operation is performed by following all those paths from the root to the leaves that are satisfied by the event. Intuitively, this data structure yields a scaleable algorithm because it exploits the commonality between subscriptions as shared prefixes of paths from root to leaf.

We assume that addition and deletion of subscriptions are rare occurrences relative to the rate of published events. We envision that changes to the subscription set are batched and periodically propagated to all brokers. Hence, we describe here the "steady state" matching algorithm that executes between changes to the set of subscriptions. We distinguish between "tree preparation time", when the subscriptions change and the PST is constructed or updated, and "matching time", when a particular event is matched using the PST data structure.

Figure 2 shows an example of a PST used for matching for an event schema consisting of five attributes a_1 through a_5. These attributes could represent, for example, the stock issue, price, or volume attributes mentioned above. The root of the tree corresponds to a test of the value of attribute a_1, the nodes at the next level correspond to a test of attribute a_2, etc. The branches are labeled with the values of the attributes being tested. In the example, we only show equality tests (other types of tests are also possible, see [2]), so the right branch of the root represents the test $a_1 = 1$. The left branch of the root, with label *, means that the subscriptions along that branch do not care about the value of the attribute. Each leaf is labeled with the identifiers of all the subscribers wishing to receive events matching the predicate, i.e., the conjunction of all tests from the root to the leaf. For example, in Figure 2, the rightmost leaf corresponds to one or more subscriptions whose predicate is (a_1=1 & a_2=2 & a_3=3 & a_5=3). Since a_4 does not appear in this subscription, it is represented by a label * in the PST.

Given this tree representation of subscriptions, the matching algorithm proceeds as follows. We begin at the root, with current attribute a_1. At any non-leaf node in the tree, we find the value v_j of the current attribute a_j. We then traverse any of the following edges that apply: (1) the edge labeled v_j if there is one, and (2) the edge labeled * if there is one. This may lead to either 0, 1, or 2 successor nodes (or more in the general case where the tests are not all strict equalities). We initiate parallel subsearches at each successor node. When any of the parallel subsearches reaches a leaf, all subscriptions at that leaf are added to the list of matched subscriptions. For example, running the matching algorithm with the matching tree of Figure 2 and the event a = <1, 2, 3, 1, 2> will visit all the nodes marked with dark circles and will match four

[1] Subscriptions are root-to-leaf since every subscription specifies a value for every attribute, and unspecified attributes are implicitly considered to be wild-cards.

subscription predicates, corresponding to the dark circles at leaf nodes.

The way in which attributes are ordered from root to leaf in the PST can be arbitrary. In our experience, however, performance seems to be better if the attributes near the root are chosen to have the fewest number of subscriptions labeled with a *.

In the companion paper [2], we have analytically shown that the cost of matching using the above algorithm increases *less* than linearly as the number of subscriptions increase.

2.1 Optimizations

A number of optimizations may be applied to the parallel search tree to decrease matching time -- these optimizations are explained fully in [2].

1. **Factoring:** Some search steps can be avoided, at the cost of increased space, by factoring out certain attributes. That is, certain attributes --- preferably those for which the subscriptions rarely contain "don't care" tests --- are selected as indices. A separate subtree is built for each possible value (or for ranges, each distinguished value range) of the index attributes.
2. **Trivial Test Elimination:** Nodes with a single child which is reached by a *-branch may be eliminated.
3. **Delayed Branching:** Traversing *-branches may be delayed until after a set of predicate tests have been applied. This optimization prunes paths from those *-branches which are inconsistent with the tests.

It is worth noting that, under certain circumstances, after applying optimizations, the parallel search tree will no longer be a tree but instead a directed acyclic graph.

3 The Link Matching Algorithm

The previous section described a non-distributed algorithm for matching events to subscriptions. This section presents the central contribution of this paper -- an extended matching algorithm for a network of brokers, and publishing and subscribing clients (as shown in Figure 3). The problem, in this case, is to efficiently deliver an event from a publisher to all distributed subscribers interested in the event.

Link matching is our strategy for multicasting events without using destination lists. After receiving an event, each broker receiving an event performs just enough matching steps to determine which of its neighbors should receive it. As shown in Figure 3, a broker is connected to its neighbors, which may be brokers or clients, via *links* (this figure shows a spanning tree derived from the actual non-tree broker network). That is, each broker, rather than determining which subset of all subscribers is to receive the event, instead computes which subset of its neighbors is to receive the event, i.e., it determines those links along which it should transmit the event. Intuitively, this approach should be more efficient because the number of links out of a broker is typically much less than the total number of subscribers in the system.

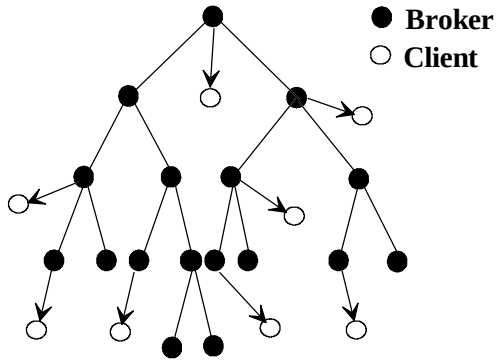


Figure 3: Spanning tree of broker network

To run link matching, each broker in the network has a copy of all the subscriptions organized into a PST data structure as described in the previous section. Each broker performs the following three steps:

1. At PST preparation time, each broker annotates each node of its PST with a vector of *trits*, each trit being a three-valued indicator with values "Yes," (Y) "No," (N) or "Maybe" (M), where the vector has one trit position per link from the given broker.
2. At PST preparation time, each broker prepares an *initialization mask* based on the network topology. There may be a different initialization mask per potential publisher. The mask also has the form of a trit vector. The mask has M's in each position corresponding to neighbors "downstream" of the given broker with respect to the spanning tree rooted at the given publisher.
3. At matching time, we run the parallel search tree algorithm of the previous section, but in addition, we refine the initialization mask as we search, changing M trits to either Y or N until all values of the mask are either Y or N.

These three steps are described in detail in the following three subsections.

3.1 Annotating the PST

A broker's subscription PST is annotated as given below in preparation for matching. The approach we describe here for computing PST annotations is limited to trees with only equality tests and *don't care* branches. A

Table 1: Alternative and Parallel Combine

Alternative	Yes	Maybe	No
Yes	Y	M	M
Maybe	M	M	M
No	M	M	N

Parallel	Yes	Maybe	No
Yes	Y	Y	Y
Maybe	Y	M	M
No	Y	M	N

more general solution requires the use of a *parallel search graph* and is not described here to conserve space.

Each broker annotates each node of the PST with a *trit vector annotation.* This annotation vector contains *m* trits, one per outgoing link from the broker. As mentioned earlier, each trit is either Y, N, or M. The value is Y when a search reaching that node is guaranteed to match a subscriber reachable through that link, N when a search reaching that node will have no subsearch leading to a subscriber reachable through that link, and M otherwise.

Annotation is a recursive process starting with the leaves of the PST, which represent the subscriptions. We begin by annotating leaf nodes in the PST with a trit vector. Recall that each leaf node corresponds to a particular predicate and a set of subscribers. For each leaf, we create a trit vector with Y's in the trit positions corresponding to the links on the path from the given broker to the subscribers associated with that leaf, and N's in all other trit positions.

After all the leaves have been annotated, we propagate the annotations back toward the root of the PST using two operators: *Alternative Combine* and *Parallel Combine*. Alternative combine is used to combine the annotations from non-* child nodes; Parallel Combine is used to merge the results of the alternative combine operations with the annotation of a child reached by a *-branch.

Intuitively, Alternative Combine takes the least specific result of two annotations. If all alternatives yield a Y, the result of Alternative Combine is a Y, and similarly with N. If the alternatives are a mixture of Y, N, and M, then the result of the Alternative Combine is M. This reflects the fact that a test will yield exactly one of the alternatives. But for Parallel Combine, both alternatives are taken; the rule is that the result is Y if either alternative is Y, otherwise the result is M is either alternative is M, else the result is N if both alternatives are N. The combine operators are shown in Table 1.

To compute a node's annotation, Alternative Combine is applied to all children of the node including the one reached through a *-branch. If no *-branch exists, one is included to represent values for which no value branch exists, and an annotation of all No values is added. Parallel Combine is then applied to this result and the *-branch.

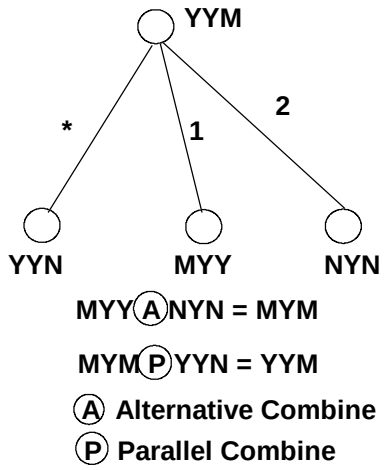


Figure 4: Computing Annotations

An example is shown in Figure 4. We assume that the child nodes have already been annotated. To compute the parent annotation, we first apply Alternative Combine to the mutually exclusive non-* outcomes. Combining MYY with NYN with Alternative Combine yields MYM. We then apply Parallel Combine to this result and the *-branch. The result of Parallel Combine applied to MYM and YYN is YYM, and this trit vector becomes the annotation on the parent node.

3.2 Computing the Initialization Mask

We assume that each broker knows the topology of the broker network as well as the best paths between each broker and each destination (i.e., subscribing client). To simplify the discussion, we ignore alternative routes for load balancing or recovery from failure and congestion. Instead, we assume that events always follow the shortest path. From this topology information, each broker constructs a *routing table* mapping each possible destination to the link which is the next hop along the best path to the destination.

We also assume that the broker knows the set of spanning trees, only one of which will ever be used by each publisher. In the case where the broker network is acyclic (Figure 3), computation of the spanning tree is straightforward. If the broker topology is not a tree, then computing the spanning tree is more complex. However, even in this case, there will be a relatively small set of different spanning trees. At worst, there will be one spanning tree for each broker that has publisher neighbors

and in most practical cases, where the broker network is "tree-like", there will be significantly fewer spanning trees.

Using these best paths and spanning trees, each broker computes the *downstream* destinations for each spanning tree. A destination is downstream from a broker when it is a descendant of the broker on the spanning tree. Based upon the above analysis, each broker associates each unique spanning tree with an *initialization mask*, one trit per link. The trit at link *l* has the value Maybe if at least one of the destinations routable via *l* is a descendant of the broker in the spanning tree; and No if none of the destinations routable via *l* are descendants of the broker[2]. The significance of the mask is that an event arriving at a broker should only be propagated along those links leading away from the publisher. Those links will begin with a mask trit of M. During matching, the M will be refined to a Y or N depending on whether a subscriber reachable along that link matches the event. But the other links will begin with a mask trit of N, indicating that the subscribers reachable over these links are not downstream from this broker. They will be reachable from some parent of this broker relative to the spanning tree from the publisher.

3.3 Matching Events

When an event originating at a publisher is received at a broker, the following steps are taken using the annotated search tree:

1. A mask is created and initialized to the initialization mask associated with the publisher's spanning tree.
2. Starting with the root node, the mask is *refined* using the trit vector annotation at the current node. During refinement, any M in the mask is replaced by the corresponding trit vector annotation. If the mask is now fully refined --- that is, it has no M trits --- then the search terminates, returning the refined mask. Otherwise, step 3 is executed.
3. The designated test is performed and, 0, 1, or 2 children are found for continuing the search as mentioned in Section 2. A subsearch is executed at each such child using a copy of the current mask. On the return of each subsearch, all Maybe trits in the current mask for which a Yes trit exists in the subsearch mask, are converted to Yes trits. After *all* the children have been searched, the remaining Maybe trits in the current mask are made No trits. The current mask is returned.
4. The top-level search terminates and sends a copy of the event to all links corresponding to Yes trits in the returned mask.

This concludes the description of the link matching algorithm.

4 Implementation and Performance

We have implemented the matching algorithms described above and tested them on a simulated network topology as well as on a real LAN, as explained in the following two subsections respectively.

4.1 Simulation Results

The goals of our simulations were twofold:

1. To measure the network loading characteristics of the link matching protocol and compare it to that of the flooding protocol.
2. To measure the processing time taken by the link matching algorithm at individual broker nodes and compare it to that of centralized matching (i.e., the non-trit matching algorithm described in Section 2).

Simulation Setup.

The simulated broker network topology is shown in Figure 5. The topology has 39 brokers and 10 subscribing clients per broker, each client with potentially multiple subscriptions. In addition, there is an unspecified number of publishing clients -- three of these publishers, shown as P1, P2, and P3 in the figure, publish events that are tracked by the simulator and the rest simply load the brokers by publishing events that take up CPU time at the brokers.

As shown in Figure 5, the 39 brokers form three regional subtrees of 13 brokers each. The roots of each of these three subtrees are connected to the roots of the other two. Also, as shown, there are a small number of lateral links between non-root nodes in the trees to allow events from some publishers to follow a different path than other publishers. This topology is intended to model a real-world wide-area network with each of the three rooted trees distributed far from each other (intercontinental), but the brokers within a tree closer to each other (interstate). The top-level brokers are modeled to have a one-way hop delay of 50ms, 65ms, and 75ms respectively, links from them to their next level neighbors is 25ms, the third level hop delay is 10ms, and the hop delay to clients is 1ms. Lateral links have a delay of 50ms. (To simplify the simulation, we assumed constant hop delays.)

[2] In some cases, where some destinations are reachable through a link downstream on some spanning trees and not on others, it may be necessary to split the physical link into two or more "virtual" links. After matching, any events "sent" along two virtual links having the same physical link can be coalesced into a single sent event.

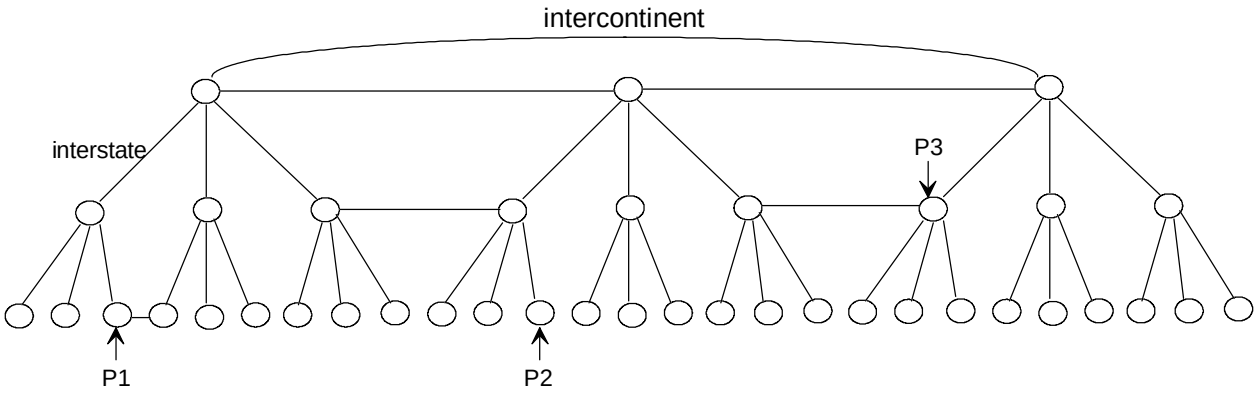


Figure 5: Simulated broker network topology

The simulated broker network implements a publish/subscribe system using the algorithms of the previous section. Simulation parameters control:

- the number of attributes in the event schema,
- the number of values per attribute,
- the number of factoring levels (i.e., the "preferred" attributes of Section 2.1),
- the number of subscriptions,
- the mean arrival rate of published events,
- the probabilities that a given subscription "cares about" a given attribute (i.e. does not have a wildcard). The most popular attribute has a "care" probability of *P*; each successive attribute has a lower "care" probability determined by a degradation factor *D*.

Subscriptions are generated randomly using the given probabilities. For non-* attributes, the values are generated according to a Zipf distribution. In addition, we simulate "locality of interest" in subscriptions by having subscribers within each regional subtree of the broker topology have similar distributions of interested values whereas subscriptions across from the other two subtrees have different distributions.

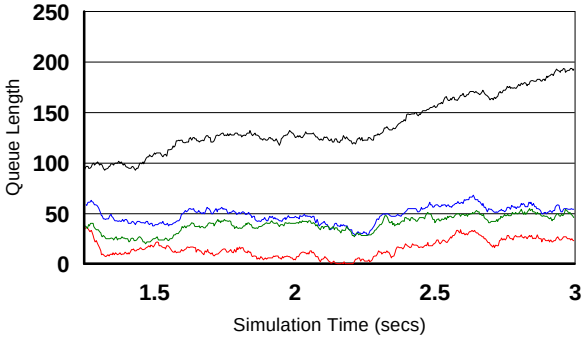
Events are also generated randomly, with attribute values in a Zipf distribution. The order of the favored values is similarly biased according to the region in which the event was published. That is, values preferred by subscribers in a region are also the values most frequently published by publishers in that region. Events are generated at the publishing brokers according to a Poisson distribution.

The simulation models passage of virtual time due to link traversal ("hop delay"), and at each broker, the queue delay at the incoming queue, the CPU consumption for executing the link matching algorithm, and the software latency for traversing the communication stack.

Network Loading Results.

As mentioned earlier, the purpose of this simulation run was to determine, for the link matching and the flooding protocols, the event publish rate at which the broker network becomes "overloaded" (or congested). A broker is overloaded when its input event queue is growing at a rate greater than the rate at which the broker can dequeue events, ultimately resulting in dropped messages. Chart 1 illustrates this phenomenon in a 1.5 second snapshot of the simulation. In this chart, the length of the queue at a particular broker is plotted for varying message publish rates (408, 417, 425, and 450 events per second per publisher), with all other parameters fixed. The graph shows that the curve for 450 events/sec is monotonically increasing, whereas the curve for 408 events/sec is occasionally draining all the events in its queue. The two curves in the middle are similar, but look ambiguous in this snapshot. In fact, one of them overloads when the simulation run is lengthened by a factor of 10, and the other drains its queue. Based on several long runs, we arrived at an overload threshold queue length of 80 (with

Chart 1: Broker Overloading



Bottom curve = 408 evts/sec; Top curve = 450 evts/sec
Middle curves = 417 & 425 evts/sec

an error bound of ± 12) for the length of simulation runs we perform. This means that with high probability, a short simulation run which reaches a queue length of 80 eventually reaches an unbounded queue length as the length of the simulation run increases, and a short run which never reaches a queue length of 80 eventually drains the queue.

The actual simulation run was performed with the following parameters. The event schema has 15 attributes (with 1 attribute used for factoring), and each attribute has 3 values. The subscriptions are generated randomly in such a way that the most popular attribute is non-* with probability P = 0.98. We control the percentage of matches by varying the degradation rate D. A smaller value of D means more "don't care" values in subscriptions, and hence less selectivity and a greater percentage of matches.

The results from the simulation run are shown in Chart 2, which shows the maximum publish rate at which the broker network does not overload, at various matching rates, for varying numbers of subscriptions. (The confidence interval for these runs is ± 5 events/sec.) For this run, the broker network is defined as overloaded when any one broker in the network has overloaded. The chart shows that the flooding protocol overloads at the same publish rate regardless of the percentage of matches or the number of subscriptions. On the other hand, the link matching protocol is able to handle much higher publish rates without overloading when each event is destined to a small percentage of subscriptions, i.e., when subscriptions are highly selective.. In the case where events are distributed quite widely, the difference is not as great, since most links will be used to distribute events in the link matching protocol. We expect that this result will be more pronounced when the broker network has a much larger number of links at each broker than the one shown in Figure 5.

This result illustrates that link matching is well-suited to the type of selective multicast that is typical of pub/sub systems deployed on a WAN.

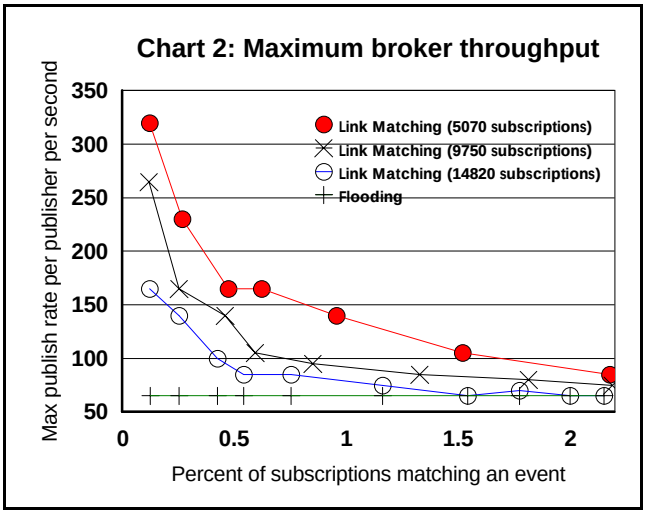
Matching Time Results.

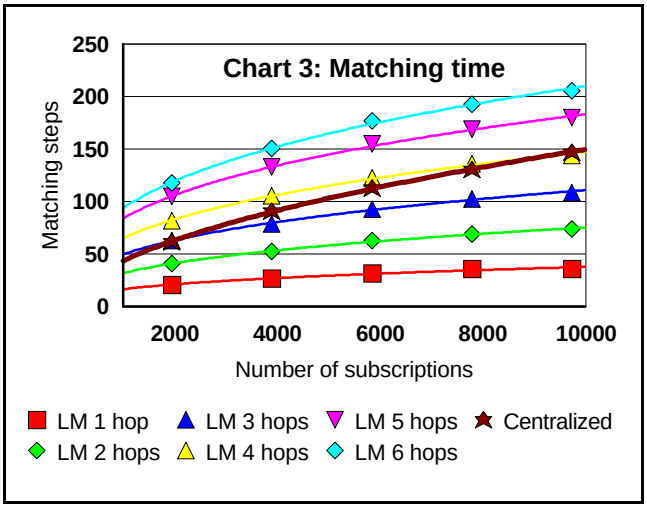
As mentioned earlier, the purpose of this simulation run was to measure the cumulative processing time taken by the link matching algorithm and the centralized (non-trit) matching algorithm. The processing time taken per event in the link matching algorithm is the sum of the times for all the partial matches at intermediate brokers along the way from publisher to subscriber.

This simulation run was performed with the following parameters. The event schema has 10 attributes (with 3 attributes used for factoring), and each attribute has 3 values. The subscriptions are generated in such a way that the first attribute is non-* with probability P = 0.98, and this probability decreases at the rate of D = 82% as we go from the first to the last attribute. Again, this means that subscriptions are very selective -- on average, each event matches only about 1.3% of subscriptions. The number of events published is 1000.

The results from the simulation run are shown in Chart 3. For the link matching algorithm, six lines, "LM 1 hop" through "LM 6 hops", are shown -- these correspond to the number of hops an event had to traverse on its way from a publishing broker to a subscriber. On the Y axis, the chart shows the number of "matching steps" performed on average. A matching step is the visitation of a single node in the matching tree. Our current implementation has traded off time efficiency in favor of space efficiency to perform a matching step in just over 100 microseconds on a Pentium 150 MHz machine. However, we estimate that a time efficient implementation can execute a matching step in the order of ten microseconds.

The chart shows that, for 10000 subscriptions, the cumulative matching steps for up to four hops using the





link matching algorithm is not more than the number of matching steps taken by the centralized algorithm. For more than four hops the link matching algorithm takes more matching steps, however the link matching protocol is still a better choice over the centralized algorithm because of several reasons:

1. The extra processing time for link matching (of the order of much less than 1ms, see the following section) is insignificant compared to network latency (of the order of tens of ms).
2. The improvement in latency from publishers to regional subscribers obtained by decentralizing brokers is significant.
3. For really large numbers of subscribers (i.e., much beyond 10000), the slopes of the lines in Chart 3 indicate that centralized matching may take more steps than link matching.

4.2 System Prototype

In order to experiment with the systemic aspects of matching and content based routing, we have also built a Java based broker network prototype. One of the purposes of the prototype was to study the cost of traversing transport layers relative to the cost of matching in commonly available implementations, such as Java's implementation of TCP/IP.

Running our prototype code, a Pentium 100Mhz machine running the centralized matching algorithm of Section 2 could match an event against 1000 subscriptions in just over half a millisecond and 25000 subscriptions in just over 4 milliseconds. (In these runs, we used a schema of 30 30-valued attributes, and subscriptions in which 65% were wild-cards.) We estimate that with top-of-the-line hardware available in 1999, a broker will be able to match an event against 25000 subscriptions in less than one millisecond using the centralized matching algorithm. With link matching, a broker may be able to achieve a factor of 4 or 5 over that (see Chart 3), yielding a throughput of up to 5000 events per second.

Therefore, the question is whether a standard transport layer implementation of, say, Java's TCP/IP, can handle this throughput. In our current broker implementation, running on a Pentium Pro 200, with 20 subscribing clients running on Pentium 100 to 133 MHz machines, we seem to peak at 700 events/second per subscriber. (In these runs, the event schema and values were trivially small.) Although this yields a throughput "across the broker" of 14000 events/sec, the cost of traversing the transport layer is unfortunately an order of magnitude slower than the matching algorithm. Better transport layer performance is part of ongoing work.

5 Related Work

As mentioned earlier, alternatives to the link matching approach were either to (1) first compute a destination list for events by matching at or near the publisher and then distributing the event using the distribution list, or (2) to multicast the event to all subscribers which would then filter the event themselves.

As mentioned in the introduction, computing a destination list (using, e.g., the matching algorithm presented in Section 2) is a good approach for small systems involving only a few subscribers, but impractical for large systems with scalability requirements.

Multicasting an event and then filtering also has its disadvantages. Lack of scalability and an inability to exploit locality was shown for the flooding approach for event distribution. Flooding is a good approximation of the broadcast approach since most WAN multicast techniques require the use of a series of routers or bridges connecting LAN links. IP multicast [5][1] allows subscriptions to a subrange of possible IP addresses known as class D addresses. Subscriptions to these groups is propagated back through the network routers implementing IP. Pragmatic General Multicast [16] has been proposed as an internet-wide multicast protocol with a higher level of service. This protocol is an extension of IP multicast that provides "TCP-like" reliability, and therefore also relies on multicast-enabled routers. A mechanism for multicast in a network of bridge-connected LANs is proposed in [7]. In this approach, members of a group periodically broadcast their membership information to an all-bridge group. Bridges note these events and update entries in a multicast table, including an expiration time.

The content-based subscription systems that have been developed do not yet address wide-area, scaleable event distribution, i.e. although they are *content-based subscription* systems, they are not *content-based routing* systems. SIENA allows content-based subscriptions to a distributed network of event servers (brokers) [6]. SIENA filters events before forwarding them on to servers or clients. However, a scaleable matching algorithm for use at each server has not been developed. The Elvin system [14] uses an approach similar to that used in SIENA. Publishers are informed of subscriptions so that they may "quench" events (not generate events) for which there are no subscribers. In [14], plans are discussed for optimizing Elvin event matching by integrating an algorithm similar to the parallel search tree. This algorithm, presented in [8], converts subscriptions into a deterministic finite automata for matching. However, no plans for optimizations for broker links (such as our optimization through trit annotation) are discussed.

Another algorithm for optimizing matching is discussed in [9]. At analysis time, one of the tests a_{ij} of each subscription is chosen as the *gating* test; the remaining tests of the subscription (if any) are *residual* tests. At matching time, each of the attributes a_j in the event being matched is examined. The event value v_j is used to select those subscriptions i whose gating tests include $a_{ij} = v_j$. The residual tests of each selected subscription are then evaluated: if any residual test fails, the subscription is not matched; if all residual tests succeed, the subscription is matched. Our parallel search tree performs this type of test for each attribute, not just a single gating test attribute.

One outlet for the work presented in this paper could be through Active Networks [17], which allow the dynamic inclusion of code (such as our matching and multicasting components) at routers.

6 Discussion and Future Work

Link matching takes the approach of replicating all the subscriptions at all the brokers in the system and (partially) matching the event at each broker. In effect, this approach trades off the complexity of maintaining a replicated subscription set and of consuming CPU time at each broker in exchange for reducing message traffic. This tradeoff is justifiable in the kind of systems we're looking at for two reasons. First, we assume that subscriptions change at a far slower rate than events published. Second, the CPU time required for link matching (much less than 1 ms for a large filter-base) is a small fraction of network latencies (tens of milliseconds).

We also assume that the configuration of the broker network is relatively static, i.e., changes to it are extremely rare. We have designed protocols for dynamically changing the configuration of the broker network; however, a description of these is outside the scope of this paper.

Two-tier Routing. An alternative approach to the link matching protocol is to match published events at the publisher and multicast (using a group) to all relevant brokers to which subscribers are connected to. Those brokers in turn deliver events to clients. However, in this case, the number of multicast groups required will be 2^B for B brokers, which may be too large for some systems. To reduce the number of groups required, events may be partially matched at the publisher, multicast to broker groups, then the remaining matching can be performed at the receiving brokers before delivering to clients. This is a promising direction for future work.

We recognize that the simulation described in this paper does not model a truly large network. We consider simulating large network topologies, especially with high fan-out at each broker, essential future work. In addition, we believe that in such networks it may be valuable to adopt a hybrid approach that varies between pure link matching and flooding depending upon the probability that a given "Maybe" trit resolves to a "Yes".

Ongoing work is concentrating on further validation of our approach to content-based routing. We are currently working to deploy our content-based routing brokers on a large private network. This will allow us to conduct system tests under actual application loads. Sample applications will include some from the financial trading and process control domains. In addition to these system tests, we are also continuing work with our simulator to examine different types of messaging loads. In particular, since many publish/subscribe applications exhibit peak activity periods, we are examining how our protocol performs with bursty event loads.

There is additional ongoing work on extending the publish/subscribe model to support multiple information spaces, and stateless and stateful event transforms to propagate events between information spaces. Some preliminary results from this work are presented in a workshop paper [3].

7 Conclusions

In this paper, we have presented a new multicast technique for content-based publish/subscribe systems known as link matching. Although several publish/subscribe systems have begun to support content-based subscription, the novel contribution of link matching is that *routing* is based on a hop-by-hop partial matching of published events. The link matching approach allows distribution of events to a large number of information consumers distributed across a WAN without placing an undue load on the network. The approach also exploits locality of subscriptions.

We evaluate how an implementation of content-based routing protocol performs, by showing that a broker network stays up while running the link matching algorithm whereas brokers get overloaded for the same event arrival rate running a flooding algorithm, since brokers have larger numbers of events to process in the flooding case. This shows that content-based routing using link matching supports a more general and flexible form of publish-subscribe, while admitting a highly efficient implementation.

Acknowledgments.

The authors wish to thank the anonymous referees for several useful comments and suggestions. We also thank Shrideep Pallickara for help with the simulation runs.

8 Bibliography

[1] Lorenzo Aguilar. "Datagram Routing for Internet Multicasting," ACM Computer Communications Review, 14(2), 1984. pp. 48-63.

[2] Marcos Aguilera, Rob Strom, Daniel Sturman, Mark Astley, and Tushar Chandra. Matching Events in a Content-based Subscription System. Proceedings of the 18th ACM Symposium on the Principles of Distributed Computing, May 1999, Atlanta, GA. (Also available from URL http://www.research.ibm.com/gryphon.)

[3] Guruduth Banavar, Marc Kaplan, Kelly Shaw, Rob Strom, Daniel Sturman, and Wei Tao, 1999. Information Flow Based Event Distribution Middleware. Proceedings of the Middleware Workshop at the International Conference on Distributed Computing Systems '99, Austin, TX.

[4] K. P. Birman. "The process group approach to reliable distributed computing," pages 36-53, Communications of the ACM, Vol. 36, No. 12, Dec. 1993.

[5] Uyless Black. TCP/IP & Related Protocols, Second Edition. McGraw-Hill, 1995. pp. 122-126.

[6] Antonio Carzaniga, *Architectures for an Event Notification Service Scalable to Wide-area Networks*. Ph.D. Thesis. Politecnico di Milano. December, 1998. Available from http://www.cs.colorado.edu/~carzanig/papers/

[7] Stephen E. Deering. "Multicast Routing in InterNetworks and Extended LANs," ACM Computer Communications Review, 18(4), 1988 . pp. 55-64.

[8] John Gough and Glenn Smith. "Efficient Recognition of Events in a Distributed System," Proceedings of ACSC-18, Adelaide, Australia, 1995.

[9] Eric N. Hanson, Moez Chaabouni, Chang-Ho Kim, Yu-Wang Wang. "A predicate Matching Algorithm for Database Rule Systems," pages 271-280, SIGMOD 1990, Atlantic City N. J., May 23-25 1990.

[10] Shivakant Mishra, Larry L. Peterson, and Richard D. Schlichting. Consul: A Communication Substrate for Fault-Tolerant Distributed Programs, Dept. of computer science, The University of Arizona, TR 91-32, Nov. 1991.

[11] Object Management Group. CORBA services: Common Object Service Specification. Technical report, Object Management Group, July 1998.

[12] Brian Oki, Manfred Pfluegl, Alex Siegel, Dale Skeen. "The Information Bus - An Architecture for Extensible Distributed Systems," pages 58-68, Operating Systems Review, Vol. 27, No. 5, Dec. 1993.

[13] David Powell (Guest editor). "Group Communication", pages 50-97, Communications of the ACM, Vol. 39, No. 4, April 1996.

[14] Bill Segall and David Arnold. "Elvin has left the building: A publish/subscribe notification service with quenching," Proceedings of AUUG97, Brisbane, Austrailia, September, 1997.

[15] Dale Skeen. Vitria's Publish-Subscribe Architecture: Publish-Subscribe Overview, http://www.vitria.com/

[16] Tony Speakman, Dino Farinacci, Steven Lin, and Alex Tweedly. "PGM Reliable Transport Protocol," IETF Internet Draft. August 24, 1998.

[17] D. Tennenhouse, J. Smith, W. D. Sincoskie, D. Wetherall, G. Minden. "A Survey of Active Network Research," *IEEE Communications Magazine*. January, 1997, Vol. 35, No. 1. pp. 80--86.