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
Contact: banavar@watson.ibm.com

Abstract. *The publish/subscribe (or pub/sub) paradigm is a simple and easy to use 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 messages. This model is considerably more flexible than subject-based pub/sub, however it was previously not known how to efficiently multicast published messages to interested content-based subscribers within a network of broker (or router) machines. This shortcoming limits the applicability of content-based pub/sub in large or geographically distributed settings. In this paper, we develop and evaluate a novel and efficient technique for multicasting within a network of brokers in a content-based subscription system, thereby showing that content-based pub/sub can be deployed in large or geographically distributed settings.*

1 Introduction

The *publish/subscribe* paradigm is a simple, easy to use and efficient to implement 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 also typically contains *message brokers* that are responsible for routing messages 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 are required to 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 [6][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 as many dimensions as event attributes, 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 removes 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 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 message broker.
- The problem of efficiently *multicasting* events within a network of message brokers. This problem becomes crucial in two settings: 1) when the pub/sub system is geographically distributed and message 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

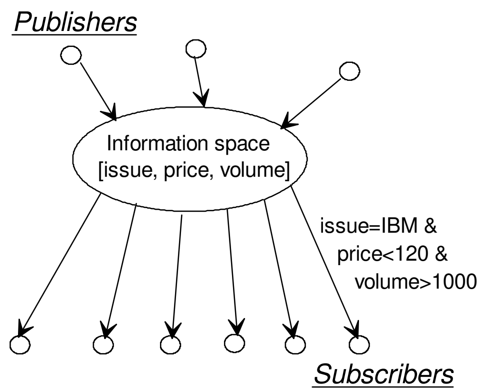


Figure 1: Pub/Sub Information Space

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 trivial to solve: the matching problem is solved using a mere 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, matching or multicasting. 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) in the *match-first* approach, 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) in the *flooding* approach, the message is broadcast or flooded to all destinations using standard multicast technology and unwanted messages are filtered out at these destinations. The match-first approach works well in small systems, but in a large system with thousands of potential destinations, the increase in message size makes the approach impractical. Further, with this approach we may have multiple copies of the same message going over the same network link on its way to multiple remote subscribers. The flooding approach suffers when, in a large system, only a small percentage of clients want any single message. Furthermore, the flooding technique cannot exploit *locality* of information requests, i.e., when clients in a single geographic area are, for many applications, likely to have similar requests for data.

The central contribution of this paper is a new protocol for *content-based routing*, an efficient solution to the multicast problem for content-based pub/sub systems. With this protocol, called *link matching*, each broker partially matches events against subscribers at each hop in the network of brokers to determine which brokers to send the message. Further, each broker forwards messages to its subscribers based on their subscriptions. The disadvantages of the match-first approach are avoided since no additional information is appended to the message headers. Further, at most one copy of a message is sent on each link. The disadvantages of the flooding approach are avoided as the message is only sent to brokers and clients needing the message, 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 present a 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 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.

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 our 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 (or PST) data structure, in which each subscription corresponds to a path from the root to a leaf. 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.

Figure 2 shows an example of a matching tree 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 (although range tests are also possible), 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., all the tests from the root to the leaf. For example, in Figure 2, the rightmost leaf corresponds to a subscription 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 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:** Following *-branches may be delayed until after a set of tests have been applied. This optimization prunes paths from that *-branch 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, publishers, and subscribers (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.

One straightforward solution to this problem is to perform the matching algorithm of the previous section at the broker nearest to the publisher, producing a destination list consisting of the matched subscribers. This destination list may be undesirably long in a large network with thousands of subscribers, and it may be infeasible to transmit and process large messages containing long destination lists throughout the network.

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, neighbors may be brokers

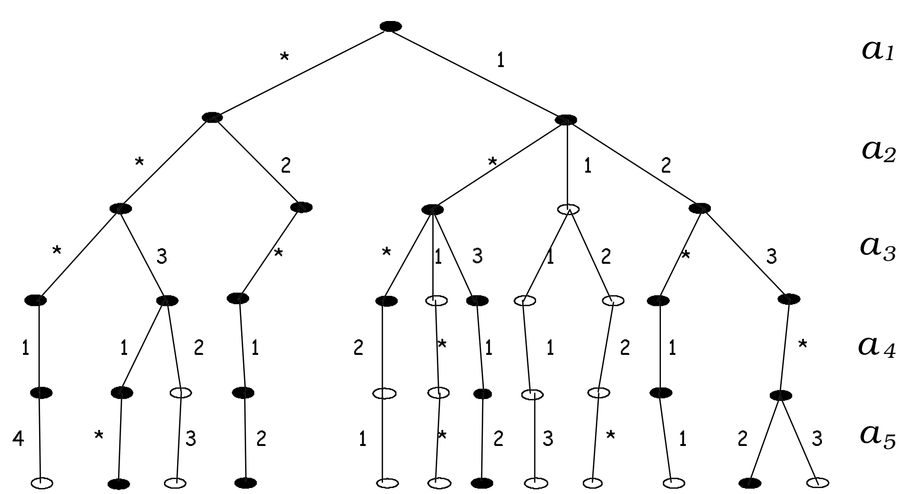


Figure 2: Matching Tree

or clients (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 message. 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.

To perform link matching, we use the parallel search tree (PST) structure of the previous section, where each path from root to leaf represents a subscription. We augment the PST with vectors of *trits*, where the value of each trit is either "Yes," (Y) "No," (N) or "Maybe" (M). We begin by *annotating* leaf nodes in the PST with a trit vector of size equal to the number of links out of that broker. For each link out of a broker, a position in a trit vector determines whether to send matched events down that link, based on whether there exists a subscription reachable via that link. Leaf annotations are then propagated to non-leaf nodes in a bottom-up manner. A "Yes" in a trit annotation means that (based on the tests performed so far) the event will be matched by some subscriber that is best reached by sending the message along the given link; "No" means that the event will definitely not be matched by any subscriber along that link; and "Maybe" means that further searching must take place to determine whether or not there is such a subscriber. Annotations are described in more detail below.

The link matching algorithm consists of the following three steps. First, at each broker, the parallel search tree is annotated with a trit vector encoding link routing information for the subscriptions in the broker network. Second, an *initialization mask* of trits must be computed at each broker for each spanning tree used for message routing. (Collectively, the masks for a single spanning tree across all the brokers encode the spanning tree in the network.) Third, at match time the initialization mask for a given spanning tree (based on the publisher) is refined until the broker can determine whether or not to send a message on each link, that is, until all values in the mask are either Yes or No. These three steps are described in detail in the following three subsections respectively.

3.1 Annotating the PST

Each broker in the network has a copy of all the subscriptions, organized into a PST as discussed in the previous section, and illustrated in Figure 2. Note that the approach we describe here for computing tree annotations is limited to trees with only equality tests and *don't care* branches. A 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

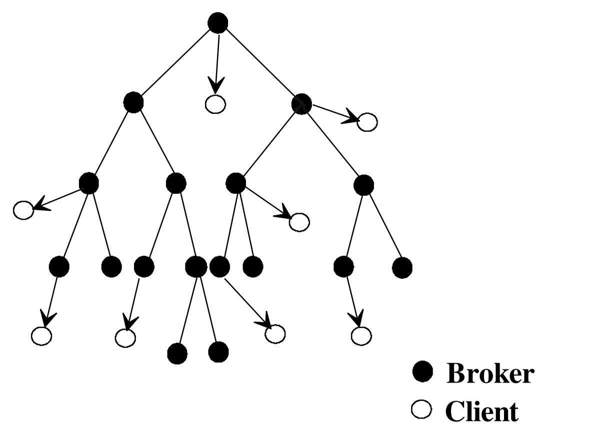


Figure 3: Broker Network

earlier, the trit is Yes when a search reaching that node is guaranteed to match a subscriber reachable through that link, No when a search reaching that node will have no subsearch leading to a subscriber reachable through that link, and Maybe otherwise.

Annotation is a recursive process starting with the leaves of the parallel search tree, which represent the subscriptions. We label each leaf node trit in link position l with Y if one of that leaf node's subscribers is located at a destination reached through link l, and N otherwise. 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.

The operators are shown in Figure 4. Intuitively, Alternative Combine takes the least specific result of two annotations. That is, Maybes dominate Yes or No results. Parallel Combine takes the most liberal result of two annotations. That is, Yes dominates Maybe; Maybe dominates No.

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.

An example is shown in Figure 5.

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. 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*

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

Figure 4: Alternative Combine and Parallel Combine

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[1]. The significance of the mask is that an event arriving at a broker should only be propagated along those links leading to descendant destinations -- that is, those links whose mask bit is M and will eventually be refined to a Y via matching, described below.

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.

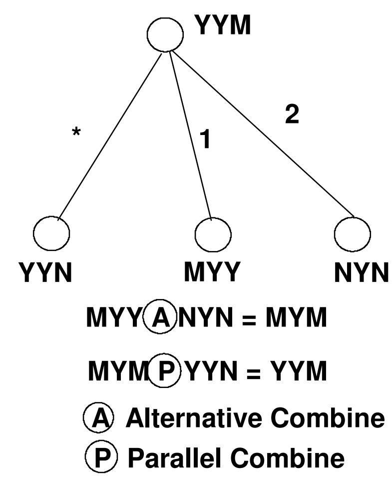


Figure 5: Computing Annotations

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).

[1] In some cases, where some destinations reachable through a link downstream on some spanning trees and are not on others, the search may be optimized by splitting the link into two or more "virtual" links.

Simulation Setup

The simulated broker network topology is shown in Figure 6. 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 messages that take up CPU time at the brokers.

As shown in Figure 6, the 39 brokers form three trees of 13 brokers each. The root of each of these three trees 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 messages 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 about 65 ms, links from them to their next level neighbors is 25ms, the third level hop delay is about 10ms, and the hop delay to clients is 1ms.

The broker network simulates an information space with several control parameters, such as the number of attributes in the event schema, the number of values per attribute and the number of factoring levels (i.e., the preferred attributes of Section 2.1). Subscriptions are generated randomly, but one of the control parameters is the probability that each attribute is a * (i.e., don't care). 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 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. Events arrive at the publishing brokers according to a Poisson distribution. The mean arrival rate of published events, which is a key parameter, is controlled by a user specified parameter.

In the simulation, time is measured in "ticks" of a virtual clock, with each tick corresponding to about 12 microseconds. The virtual clock, used only for simulation purposes, is implemented as synchronized brokers' clocks. Each event carries with it its "current" virtual time from the beginning of the simulation. An event spends time traversing a link ("hop delay"), waiting at an incoming broker queue, getting matched, and being sent (software latency of 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), for a varying number of subscriptions. A broker is overloaded when its input message queue is growing at a rate higher than the broker processor can handle.

This simulation run was performed with the following parameters. The event schema has 10 attributes (with 2 attributes used for factoring), and each attribute has 5 values. The subscriptions are generated randomly in such a way that the first attribute is non-* with probability 0.98, and this probability decreases at the rate of 85% as we go from the first to the last attribute. This means that subscriptions are very selective -- on average, each event matches only about 0.1% of subscriptions. The number of events published is 500.

The results from the simulation run are shown in Chart 1. The chart shows that a broker network running the flooding protocol saturates at significantly lower event publish rates than the link matching protocol for any number of subscriptions. In particular, when each event is destined to only a small percentage of all clients, link matching dramatically outperforms flooding. In the case where events are distributed quite widely, the difference is not as great, since most links are used to distribute events in the link matching protocol. 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.

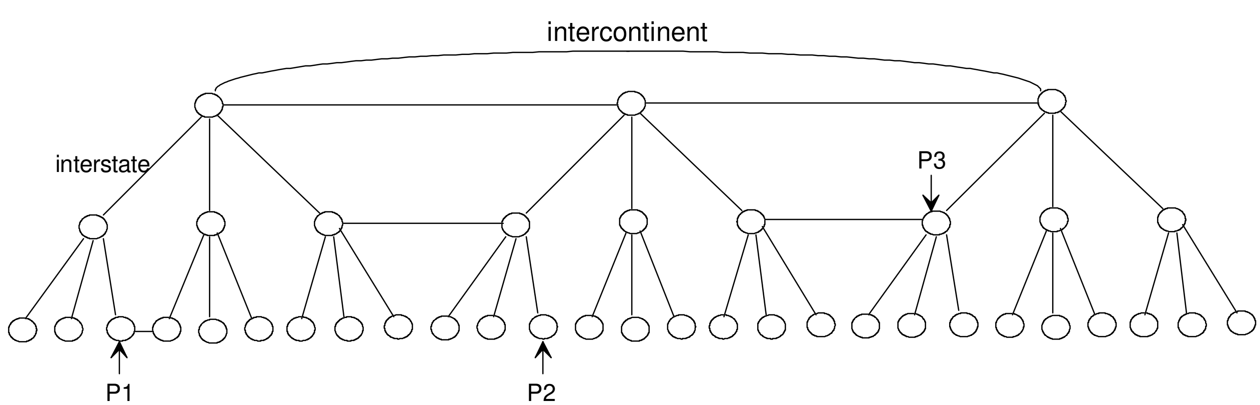
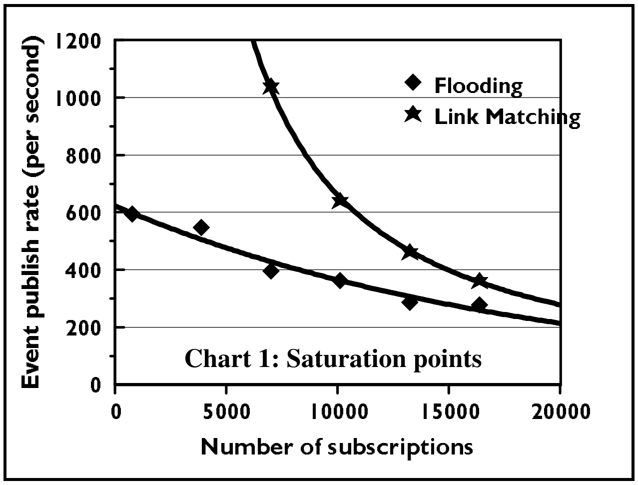
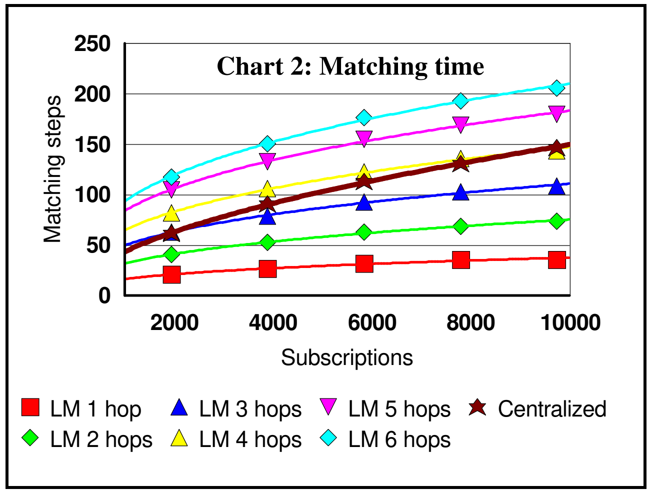


Figure 6: Simulated broker network topology





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 randomly in such a way that the first attribute is non-* with probability 0.98, and this probability decreases at the rate of 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 2. 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. Although our current implementation has traded off time efficiency in favor of space efficiency, we estimate that a time efficient implementation can execute a matching step in the order of a few microseconds.

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

4.2 System Prototype

We have implemented the matching algorithms in a network of broker nodes where brokers are connected using a specified topology. A broker network may implement multiple information spaces by specifying an event schema (one per information space) defining the type of information contained in each event. Clients subscribe to an information space by first connecting to a broker node, then providing subscription information which includes a predicate expression of event attributes. This section describes the implementation of such a broker node.

As illustrated in Figure 7, each broker node consists of a matching engine, client and broker protocols, a connection manager and a transport layer. The matching engine which implements one of the matching algorithms described earlier, consists of a subscription manager, and an event parser. A subscription manager receives a subscription from a client, parses the subscription expression, and adds the subscription to the matching tree. An event parser first parses a received event, then un-marshals it according to the pre-defined event schema. The matchine engine then uses the implemented matching algorithm to get a list of subscibers interested in the un-marshaled event.

The broker to client protocol is implemented by the client protocol object, whereas the broker to broker protocol is implemented using the broker protocol object. These protocol objects are robust enough to handle transient failures of connections by maintaining an event log per client. Once a client re-connects after a failure, the client protocol object delivers the events received while the client was dis-connected. A garbage collector periodically cleans up the log. The connection manager object maintains the connections to clients and the other brokers in the network.

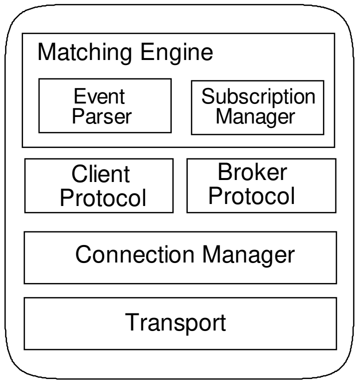


Figure 7: Broker Components

The transport layer sends and receives messages to and from clients and other brokers in the network. To improve scalability, it implements an asynchronous "send" operation by maintaining a set of outgoing queues, one per connection. A broker thread sends a message by en-queueing it in the appropriate queue. A pool of sending threads is responsible for monitoring these queueues for outgoing messages, and sending them to destinations using the underlying network protocol.

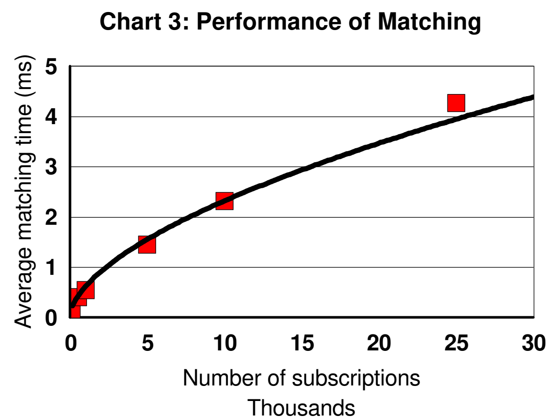
Currently, broker nodes are implemented in Java using TCP/IP as the network protocol. In an experimental setup where a 200 MHz pentium pro PC is used as a broker node, and low end PCs (using 133 MHz pentium processors) are used as clients connected using a 16MB token ring network, the current implementation of the broker can deliver upto 14,000 events/sec. Also, as shown in Chart 3 for the pure matching algorithm, brokers can perform matching very quickly, at the rate of about 4ms for 25,000 subscribers. In fact, our matching algorithms are so efficient that the transport system and network costs of a broker outweigh the cost of matching at a broker.

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.

Computing a destination list is a good approach for small systems involving only a few subscribers. For these cases, the matching algorithm presented in section 2 provides a good solution. However, scalability is essential if content-based systems are to fill the same infrastructure requirements as subject-based publish/subscribe systems. In cases where destination lists may grow to include hundreds or thousands of destinations, the match-first approach becomes impractical.

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 is also reliant 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 to an all-bridge group their membership in a multicast group. Bridges note these messages 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]. Active Networks have been touted as a mechanism for eliminating the strong dependence of route architectures on Internet standards. Active Networks allow the dynamic inclusion of code either at routers or by replacing passive packets with active code. The SwitchWare project [3] follows the former approach and is most appropriate to the type of router customization proposed in this paper. With SwitchWare, digitally signed type-checked modules may be loaded into network routers. Our matching and multicasting component could be one such module.

6 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 undo 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 the flooding algorithm, since brokers have larger numbers of events to process in the flooding case. We also describe a broker implementation that can handle message loads of up to 14000 events per second on a 200 MHz Pentium PC. 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.

Future 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 message loads.

7 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, Tushar Chandra. 1998. Matching Events in a Content-Based Subscription System. Upcoming IBM Technical Report, available from http://www.research.ibm.com/gryphon.

[3] D. Scott Alexander *et al.*, "The SwitchWare Active Network Architecture," *IEEE Network Special Issue on Active and Controllable Networks,* July 1998, Vol. 12, No. 3. pp. 29--36.

[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, David S. Rosenblum, and Alexander L. Wolf. "Design of a Scalable Event Notification Service: Interface and Architecture," unpublished. Available from http://www.cs.colorado.edu/users/carzaniga/siena/index.html

[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.