Combining Flexibility and Scalability in a Peer-to-Peer Publish/Subscribe System

Chi Zhang[1], Arvind Krishnamurthy[2], Randolph Y. Wang[1], and Jaswinder Pal Singh[1]

[1] Princeton University
[2] Yale University

Abstract. The content-based publish/subscribe model has been adopted by many services to deliver data between distributed users based on application-specific semantics. Two key issues in such systems, the semantic expressiveness of content matching and the scalability of the matching mechanism, are often found to be in conflict due to the complexity associated with content matching. In this paper, we present a novel content-based publish/subscribe architecture based on peer-to-peer matching trees. The system achieves scalability by partitioning the responsibility of event matching to self-organized peers while allowing customizable matching functionalities. Experimental results using a variety of real world datasets demonstrate the scalability and flexibility of the system.
Keywords: publish/subscribe, matching, peer-to-peer

1 Introduction

The deployment and application of event-based publish/subscribe services has increased considerably over the past years. A number of emerging applications, ranging from simple personal tools to large-scale and critical systems, benefit from this paradigm. Examples include stock quote notification, Internet news feeds, real-time traffic control, and various monitoring/management systems. Publish/subscribe systems deliver events from publishers to subscribers based on their interests. Publishers and subscribers can be completely unaware of one another and communicate via the message brokers that match events to interested data users. This decoupling provides an attractive communication mechanism for building large scale distributed systems.

The expressiveness of subscriber interests is a key factor in such middlewares. Early publish/subscribe systems like TIBCO [20] and CORBA event channels [13] are subject-based. Subscribers join a set of subject groups that they are interested in and receive all messages associated with the subjects.

Content-based publish/subscribe systems allow more flexibility in specifying subscriber interests. Subscriptions specify filters on event contents. Only those events with attributes matching the filters are delivered to the subscriber. A typical application is stock quote notification. The events carry attributes of prices and trade volumes of individual stocks. Subscribers may specify triggering ranges of price or volume for the stocks that they are interested in. They get notification once events matching their subscriptions occur. Another scenario is literature reference tracking. Researchers may subscribe to new publications matching certain keywords in their titles, abstracts or bodies. They may also choose to track new papers from certain authors or citing certain previous works. In both examples, content-based filtering provides fine-grained control on the relevance of messages.

However, the power of expressiveness introduces an additional cost of matching events to the complex filters specified by subscribers. As the system scales with the number of subscriptions and the volume of event messages, a centralized matching solution cannot meet the computation and communication requirements. Therefore, we seek a solution to the scalability issue by distributing the matching responsibility to many machines. In particular, we leverage peer-to-peer overlay techniques to build a highly scalable publish/subscribe system. In our system, broker nodes self-organize and maintain a decentralized data structure that stores the subscriptions, match the events to the subscriptions, and deliver the events to relevant subscribers. Broker nodes may be added to or removed from the system without global coordination. A key problem facing such a scalable system is how to partition the workload among participating peers in a load-balanced fashion.

The flexibility provided by content expressiveness creates challenges to system scalability. While a subject-based publish/subscribe system can easily partition the workload of event delivery to a large set of servers by hashing the subjects among the servers, content-based systems have more complex subscription structures that impede the workload partition. Three factors contribute to this difficulty:

1. **High dimensionality of the content space**: a general publish/subscribe system might have to operate in a setting that involves a large number of attributes. To make things even worse, subscribers and publishers do not always speak the same schema. Subscribers seldom know in advance the schemas used by (potentially many) publishers. Even if they do, they might be interested in only a subset of it.
2. **Type flexibility**: attributes may have various types that require different filtering tests.
3. **Skewed data distribution** is common in real world subscriptions and events. It can create a load imbalance in the system that throttles the scalability.

Previous work on workload partitioning usually impose restrictions on the flexibility of subscriptions and events. In [22] and [19], the set of attributes and their values are hashed to decide the servers managing the subscriptions. This requires events and subscriptions to follow certain pre-defined schemas, and only works well with equality tests. It is difficult to efficiently support range subscriptions in such systems. Meghdoot [9] leverages CAN [15] to partition the multi-attribute space. Though it can support range subscriptions, it is still confined to numerical attributes and also can not handle skewed distributions efficiently.

Our Solution

In this paper, we propose a peer-to-peer architecture that achieves high scalability and generality. We address the expressiveness problem with a modular matching tree structure. This tree organizes the subscriptions into hierarchical groups based on their similarity. It supports flexible schemas and multiple attribute types in subscriptions and events, and allows customization of new attributes and filtering types. We distribute this matching tree in a peer-to-peer system where each peer processor manages a small fragment of the tree. They maintain the distributed tree by peer-wise communications without global coordination.

Events can enter the system from any processor. A decentralized tree navigation algorithm is used to forward the events to those tree fragments that may contain matching subscriptions. In experiments using several real world data sets, the proposed system demonstrates excellent scalability: the distributed event matching only visits a small number of processors, processors maintain a small amount of state about peers, and the workload is well-balanced across the processor set.

The next section gives a survey of related work. Section 3 details the structure of the matching tree. Section 4 discusses how the tree is distributed and how to navigate the tree in a decentralized manner. Section 5 focuses on how the distributed tree is maintained in the face of churn and changing load conditions. Section 6 presents experimental results.

2 Related Work

Several centralized algorithms for content-based publish-subscribe [8, 7, 2, 10] have been proposed to address the efficiency of the matching operation. Our matching tree bears some similarity to previous work, such as [2, 10], which also use search tree structures. The key differences are: 1) Our matching tree is more flexible, partitioning the subscriptions by both schema content and attribute value, while [2, 10] only partition by the attribute value specified in subscriptions. 2) We distribute the matching tree amongst peer processors to address the scalability problem.

Distributed content-based publish/subscribe systems deploy a network of broker servers to efficiently match and deliver events. Examples include Elvin [17], Siena [4], and Gryphon [2]. Elvin uses a central server to store subscriptions and match events. Therefore, it still imposes a bottleneck at the matching engine. Siena and Gryphon distribute the responsibility of matching events to a set of distributed servers. Events follow a multicast tree to reach all matching subscribers. However, they require the subscriptions to be replicated on all servers. This causes a burden on server management and is a stumbling block to scalability.

To address this scalability problem, several systems consider the partitioning of content-space and the subscription set. Riabov *et al.* have proposed clustering algorithms that partition similar subscriptions into multicast groups. EDN [22] partitions the content space subject to the restriction that the schema is fixed. For equality test, the attribute IDs and values are hashed to generate a key to locate the server managing it. For inequality tests, EDN uses an R-tree to decide offline how to assign subscriptions to processors, and requires each processor to maintain a complete map of this assignment. This approach is limited to small-scale systems with a fixed set of subscriptions, and it is also unclear as to whether it works efficiently for high dimensional content space.

Peer-to-peer overlays have emerged as a promising approach to realizing highly scalable distributed systems. Several systems provide application-level multicast [12, 3] that divides the data dissemination responsibilities amongst peers. They do not, however, address the selective delivery of events. Recently, Distributed Hash Tables (DHTs) have been employed to build scalable publish/subscribe systems. Scribe [5] uses Pastry [16] to build a subject-based publish/subscribe service. It hashes each topic to a peer, which then acts as the rendezvous point. The routing paths from subscribers to the rendezvous point form a multicast tree for this subject. This approach, however, can not be adapted to efficiently support the content-based publish/subscribe model.

A few previous projects have addressed content-based publish/subscribe in peer-to-peer systems. [19] partitions the content-space by hashing a set of selected attributes and their values into peer processors. The domain of attribute values are partitioned into intervals for the hashing. A range subscription may need to be decomposed to multiple intervals, resulting in storage and matching inefficiency. Furthermore, the subscriptions and events are limited by the pre-selected attribute sets. Meghdoot [9] relaxes the restrictions on subscriptions. It uses CAN [15] to manage the multi-attribute content-space. A subscription defines a rectangular region in the D-attribute content space bounded by the minimal and

maximal value specified. Unspecified attributes take the whole value range. The hyper-rectangle is projected to a point in a $2D$-dimension CAN constructed from the minimal and maximal values of the D-dimension rectangle. An event is then mapped to a rectangle in the $2D$ space, and the mapping is performed in a manner such that the rectangle covers all subscription points relevant to the event. This novel approach reduces the subscription matching problem into a range query operation in CAN. The drawback with this approach is that subscriptions are limited to numerical comparisons. Other tests like keyword subset can not be supported. Furthermore, the subscriptions are only mapped to the upper-left side of the diagonal hyper-plane of the CAN space, which may create load imbalance.

3 Content-based Event Matching

In this section, we start by describing the specification of events and subscriptions in our system. We then present the main data structure, the matching tree, used in the system.

We also note that we focus primarily on the logical organization and navigation of the matching tree in this section. The distributed operation and maintenance of the tree will be presented in following sections.

3.1 Content-based Publish/Subscribe Model

We adopt a general event-space model with multiple attributes, based on the models used in previous systems [7, 4, 2]. The contents of an event message is represented by a set of attribute-value pairs. Each attribute has a unique name or ID. We support several types of attributes: *numerical* (integer, floating point, and date/time), *string*, and *set*. The event message can be represented as $e = \{A_1 = v_1, A_2 = v_2, \ldots, A_k = v_k\}$. Events from different publishers may use different schemas, but we assume a consistent assignment of unique attribute IDs and their types across the publishers to avoid naming confusion. One could also employ hierarchical namespaces to achieve this coordination.

As an example, consider an event from a research reference database. Its contents may be formulated as $[title = \text{TTT}, date = \text{YY/MM}, authors = \{A, B, C\}, references = \{D_1, D_2, ...D_n\}]$, where *title* has *string* type, *date* is *numerical*, and *authors* and *references* fields are both of type *set*, meaning they include an unordered list of keys.

A subscription is a conjunction of predicates over the attributes. Each predicate specifies a boolean test over an attribute. The test specified by a predicate depends on the type of the attribute. Table 1 lists the type of tests supported in our system. Disjunction of predicates can be expressed by the "OR" of multiple conjunctions, so we treat a disjunctive subscription as a set of independent conjunctive subscriptions.

We do not require events and subscriptions to use the same schemas. There may be a large number of possible attributes, while any event and subscription may specify only a subset of attributes. An event matches a subscription if every predicate specified is satisfied by the attribute-value content of the event message. Not all attributes in the event need to appear in the matching subscription. The additional attributes do not affect the matching results, since the subscription does not care about the values of these attributes. However, the event does not match a subscription if an attribute specified in the subscription's predicates is missing from the event. This semi-structured matching capability is important for environments with heterogeneous publishers. Some systems, like EDN [22], require all events to use the same schema. Such restrictions limit the generality of the system and thus is not desirable.

type	tests
$Numerical$	$=, <, \leq, >, \geq$
$String$	$=, <, \leq, >, \geq$, prefix match
Set	$\ni, \supseteq$

Table 1. Predicates supported in the system

3.2 Content-Space Partition with a Matching Tree

We propose a matching tree algorithm to partition a general event space. A hierarchical tree structure is used to partition the set of subscriptions based on their predicates. Each internal node partitions the subscriptions by a similarity test, so similar subscriptions can be grouped to the same tree branch. In order to adapt to flexible attribute sets and schemas, we build the similarity tests dynamically.

Two types of similarities are used in the tests. The first is the similarity of the attribute set. The test takes an attribute from the subscriptions and hashes its name. The subscriptions are assigned to one of two branches based on the hash value. After recursive partitioning with several levels of internal nodes, each branch will have subscriptions sharing the same attribute. The second type groups subscriptions having similar value constraints for a common attribute. Depending on the type of this attribute, the test assigns the subscriptions to two branches. For convenience, we label the child branches of an internal node *L* and *R*. In addition, there is a wildcard branch, labeled as *, for subscriptions that do not contain the attribute specified by the internal node.

- - - > navigation path of event {date=05/05, authors={Y, Z}}

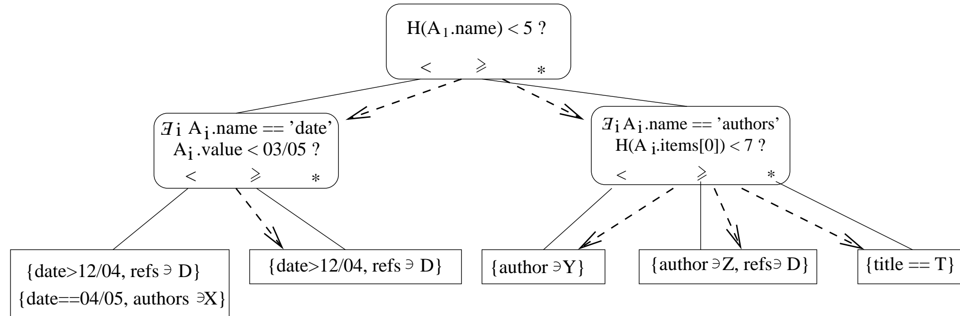


Fig. 1. Matching Tree

Figure 1 gives an example of the matching tree used for subscriptions to research publications. The root node partitions the subscriptions based on attributes specified in their predicates. It takes the first attribute in the subscription (A_1), hashes the name ($A_1.name$), and assigns the subscription to one of two branches based on the demarcating value of 5 for the result of the hash. The left child node of the root further partitions the subscriptions based on the value of the $date$ attribute. If a subscription has a predicate that tests the $date$ attribute, then it is stored in one or both of the *L* and *R* branches. For instance, if the range of the predicate on the 'date' attribute intersects with the range $(0, 03/05)$, the subscription would be inserted in the left branch; if it intersects with the range $[03/05, \infty)$, it would be inserted in the right branch; and a subscription that covers a broad range, like $\{date > 12/04, authors \ni X\}$, would be inserted in both branches. If a subscription's first attribute hashes to a value less than 5 and if that subscription does not have any pred-

icates referring to the $date$ attribute, then it is stored in the wild-card * branch. The right child of the root node partitions the subscriptions based on how they test the $authors$ attribute. Since $authors$ is a set attribute, we pick any of the keys specified in the predicate testing the $authors$ attribute, and hash it to decide the branch the subscription belongs to. The subscription $\{title == T\}$ falls into the default branch $*$, since it does not contain any predicates testing the $authors$ attribute.

Event messages also navigate the same matching tree to find matching subscriptions. Figure 1 gives an example of how an event is handled. The event starts from the root node. It is passed on to both branches, because the attributes in the event, $date$ and $authors$, hash to the *L* and *R* branches respectively. The event is further propagated through the *R* branch at the left child node based on its $date$ value. At the right child node, both *L* and *R* branches are followed, because the elements in the $authors$ field hash to either side of the pivot value 7. At the leaf nodes, a centralized matching algorithm like the counting algorithm [7] is used to match the event to the set of matching subscriptions.

Next, we give further details regarding the two partitioning methods.

3.3 Partitioning the Attribute Set

The first type of partitioning tries to group together subscriptions that test similar attributes. We first order the predicates of a subscription based on their selectivity. For simplicity, we order equality tests before subset tests, and consider inequalities as the least selective. More sophisticated techniques that take into account data distribution to order predicates regarding their selectivity are also possible. We then take the most selective predicate in the subscriptions, and hash the attribute name into a bin $H(A_1.name)$. Each child branch manage a sequence of hash bins and the subscriptions falling into the sequence. A pivot value separates the hash bins of the left and right branches.

While a subscription only descends into either the left or the right branch of this internal node, an event may follow both branches. Given an event $\{A_1 = v_1, A_2 = v_2, \ldots, A_k = v_k\}$, the left branch is taken if any of the hash values $H(A_i.name)$ corresponds to the bins on the left side of the pivot. Similarly, the right branch is taken if any of the hash values corresponds to right-hand side bins. In general, when this form of partitioning is performed iteratively at multiple internal nodes, an event with k attributes navigates into at most k branches under attribute set partitioning.

Given a set of subscriptions in a leaf node, we choose the pivot value that evenly partitions the subscriptions. When the subscriptions' most selective attribute is the same, either because of user subscription pattern or due to prior partitioning of the attribute set, we partition based on the second and third most selective attributes. Therefore, the state information maintained in an attributed set partitioning node includes the order of the attribute being hashed, the range of hash bins owned by this node, and the pivot value used for partitioning.

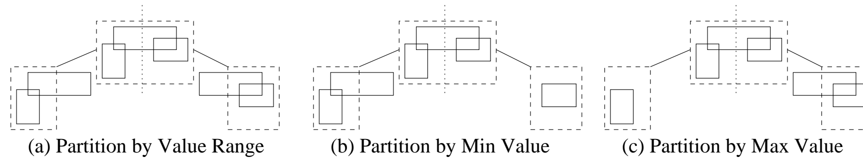
3.4 Partitioning Attribute Content

After partitioning the attribute set, each branch of the matching tree contains subscriptions with similar attributes. We can therefore partition further using the value ranges of their common attributes. We apply different strategies based on the attribute's data type.

– **Value range partition** applies to numerical attributes. It splits the value range of the attribute by a pivot value. The value range specified by predicates in the subscriptions

are compared to the pivot. If the whole range falls to the left/right of the pivot, the subscription is assigned to the left/right branch. Otherwise, the subscription is replicated into both branches. This strategy is therefore suitable for subscriptions specifying narrow value ranges, for example, equality tests. The attribute set partitioning policy that gives priority to highly selective predicates also improves efficiency of value range partition. While subscriptions may be replicated in both branches, an event only descends into one of them. So this approach reduces matching cost by using additional storage.

- **Min/max partition** divides the set of subscriptions instead of the value space. The minimal/maximal value in the constraints is used to decide the branch it belongs to. Therefore, a subscription is only assigned to one of the left/right branches. Consequently, an event may need to navigate into both branches to locate matching subscriptions. Figure 2 illustrates differences between the three strategies used to partition range constraints on a numerical attribute.
- **String value partition** is similar to value range partitioning. A subscription with a prefix predicate may be assigned to both branches if the prefix includes the pivot string.
- **Set partition** hashes the keys specified in the subscriptions and divides the hashed key space into two halves across a pivot key. A subscription specifying several keys for the set attribute may choose to follow the branch decided by any of the keys. An event message would have to navigate into all branches that its set members hash to. This is necessary to ensure that all related subscriptions can be reached. Therefore, an event message specifying k keys for the set attribute may navigate into up to k branches under multiple levels of set partitioning.



(a) Partition by Value Range (b) Partition by Min Value (c) Partition by Max Value

Fig. 2. Partitioning options based on a numerical attribute.

In all of the above mentioned types of attribute content based partitioning, the default $*$ branch may be taken if a subscription does not specify the attribute. An event always traverses into the $*$ branch if it exists, unless the attribute being partitioned is the only one specified in the event.

3.5 Choosing Partition Method

The matching tree grows by splitting leaf nodes. We aim at distributing the subscriptions in the leaf node evenly to the branches of the newly formed internal node. The two partitioning methods described above have different levels of effectiveness under different situations. When the subscriptions carry sets of attributes that differ significantly, partitioning the value space of any single attribute may only work on a small part of subscriptions while leaving the majority in the wildcard branch. Attribute set partitioning is more effective in this case. After subscriptions with the same attributes are grouped together, partitioning the content of this attribute will yield more balanced results.

When a leaf node needs to be partitioned, we scan the subscriptions in the node, and count the number of subscriptions associated with each attribute. We try to partition the

attributes that appear in at least half of the subscriptions, and choose the partition method that yields best load balance, defined as the largest number of subscriptions in the branches after split. If such attributes do not exist, we partition the attribute set.

Besides the partitioning approaches discussed above, we also use a special "partition" method that replicates the set of subscriptions to both children branches. An event may choose to follow any of the mirrored branches. As the branches are assigned to different processors, this replication spreads out the load of event matching. We use this method when the processor managing the leaf node is saturated by the event traffic targeting the leaf node. Such event hot spots may be found in some subscriptions that match a broad range of events, for example, $\{Volume \geq P_1\}$ in stock quote notification service (Section 6.1).

3.6 Extensibility

The above discussion illustrates that several different partitioning methods are used in our system. Generally, for each data type, the system needs at least one partitioning method to decide how the subscriptions and the events navigate the matching tree. Each partitioning method is implemented as a module that provides three interface functions:

- *Subscription branching*: given the state in the node, decide which branch(es) a new subscription needs to take.
- *Event branching*: given the state in the node, decide which branch(es) an event message needs to take.
- *Node split*: given the set of subscriptions in a leaf node, decide the best way to partition the subscriptions once the leaf node gets overloaded.

This modular design allows new data and predicate types to be introduced into our system, therefore ensuring generality.

4 Peer-to-Peer Matching Tree with Brushwood

In this section, we present the design of our peer-to-peer architecture. We distribute the matching tree using peer-to-peer overlay techniques in order to achieve the following:

- **Balanced distribution:** We partition the matching tree into a set of subtrees, so that the workload of managing subscriptions and matching events can be divided among peer processors in a balanced manner.
- **Locality and ability to support complex event filtering:** Since the distribution is at the granularity of subtrees, related subscriptions are stored on the same processor. Furthermore, the generality of the matching tree ensures that our system can handle subscriptions with range predicates and efficiently match events to such subscriptions.
- **Symmetric distribution that avoids hotspots:** We ensure that no processor in the system is subject to inordinately high load. We avoid distribution schemes that assign the root of the matching tree to a single processor, which is then subject to handling every new event or subscription. Instead, we make all subtrees self-contained and independent. Each processor maintains the path from the root of the matching tree to the root of the subtree in addition to maintaining the full set of internal nodes and leaf nodes of the subtree. An event or subscription could be routed to any one of the processors, which can either handle it locally or forward it to the appropriate processor(s).

– **Scalability:** We require that processors maintain small amounts of state regarding the current state of the system. In particular, each processor in our system keeps track of a logarithmic number of peers in the system. Peers periodically exchange information regarding their portion of the matching tree, so that they can maintain a weakly consistent partial view of the global matching tree. This partial view allows the processors to forward subscriptions and event messages to relevant matching tree nodes.

4.1 Brushwood

We extend the Brushwood framework described in our position paper [24] to build the peer-to-peer matching tree. Brushwood is a peer-to-peer search tree designed for scalable indexing of high dimensional data. Here we adapt its distributed organization for the publish-subscribe needs.

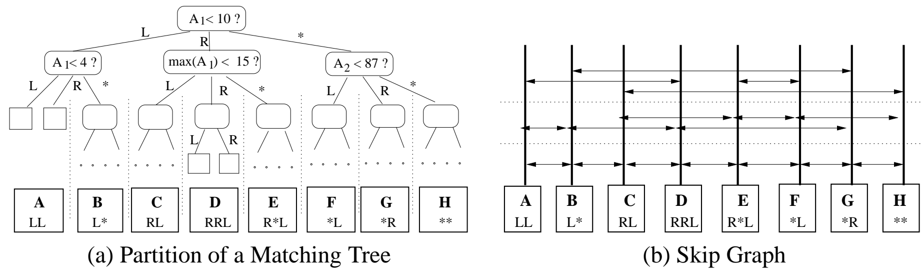
Tree distribution: Brushwood partitions a search tree into self-contained fragments cooperatively managing the distributed tree. Figure 3 (a) illustrates our approach in distributing a matching tree. The edges are labeled as 'L', 'R' and '*' for left, right and default branches. We linearize the tree nodes by pre-order traversal and then partition them into eight fragments separated by the dotted vertical bars. This partitioning method preserves locality of similar subscriptions since the low level subtrees are not split. The tree fragments are assigned to eight processors A - H, shown as the rectangles below the tree. We identify the fragments, and the processors managing them, with its *left boundary*. The left boundary is defined as the the left-most tree node in the partition under pre-order traversal. This boundary can be uniquely identified by the sequence of edge labels along the path from the root of the matching tree to the boundary node. We use this sequence as the *Tree ID* of the tree fragment. The Tree ID of each of the fragments are shown in the processor rectangles.

Data structure maintained by each processor: In a dynamic peer-to-peer system, processor joins and departures are frequent events. Each join/departure changes the location of some subtree. Therefore, we can not afford to replicate across all processors the global map of which processor owns which portion of the tree. Instead, a processor only maintains a *partial tree view*, which is a sub-graph of the global matching tree. This partial tree of a processor consists of the following: 1) all the leaf nodes managed by the processor, 2) the left boundary nodes of some selected peer processors, and 3) all internal tree nodes along the paths from the root of the matching tree to the nodes specified above in (1) and (2). Information about the peer boundary nodes are collected by contacting peer processors. The construction of the partial view is, therefore, a localized operation with cost proportional to the number of peers. The selection of peer processors is discussed later in this section. Figure 4 shows the partial view of A and D.

Event Handling: When a new event is received by a processor, the event is processed using the partial tree view. The event is propagated through the partial tree view, starting from the root of the partial tree, to determine which portions of the tree are related to the event. During this process, one or more of the following types of actions are performed:

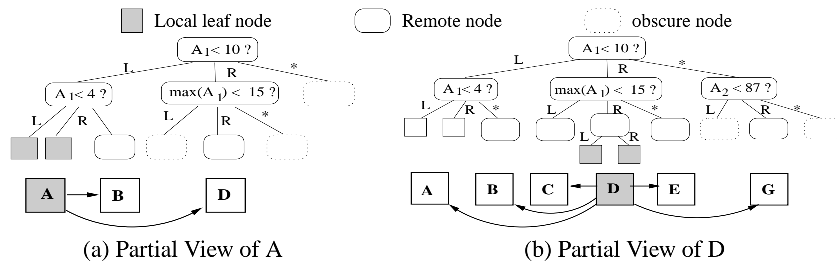
– The event is relevant to one or more of the *local* leaf nodes managed by this processor. The matching can be then performed locally.

– The event needs to be routed to a *remote* leaf node managed by a peer.
– The event is relevant to some *obscure* nodes corresponding to unknown portions of the matching tree that is not managed by any peer. The event is then routed to some peer that is more likely to be aware of the obscure node.



(a) Partition of a Matching Tree (b) Skip Graph

Fig. 3. Peer-to-peer Matching Tree



(a) Partial View of A (b) Partial View of D

Fig. 4. Partial Tree Views from Processor A and D

Example: Now we show how to perform event matching in a distributed tree with an example event message $\{A_1 = 20, A_2 = 90\}$. Assume the event enters the system from processor A. A navigates its partial tree to find all subtrees that may contain subscriptions matching this event. In this case, subtrees RR, $R*$ and $*$ are involved. A forwards the query to the processors managing these regions. RR is managed by peer D. Obscure nodes $R*$ and $*$ have to be reached by overlay routing. We route the messages to the peer that is farthest in the same direction as the obscure node (given the pre-order linearization of tree nodes) without passing over the target. In this example, all three subtrees are forwarded to peer D for further matching. D further navigates its partial tree to identify related regions to be searched. It performs local matching in subtree RR, and forwards the message to E and G for further matching. Event matching is therefore performed starting from any processor by "jumping" among the processors instead of traversing a distributed tree path from the root to the target. Each forwarding step refines the subtrees that need to be searched. The number of hops is logarithmic in the number of processors, regardless of tree depth. Subscription insertion follows a similar procedure.

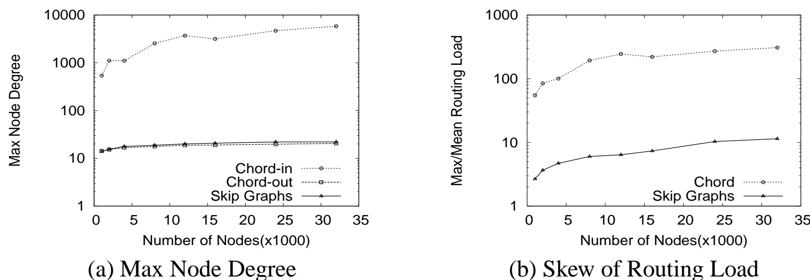
4.2 Routing Substrate

We now consider the question of establishing peers. To ensure system scalability, we limit the amount of state information managed by individual processors. Each processor only

maintains $\log N$ peers and their partition boundaries in an N-processor system. Therefore, each node join and departure can be handled efficiently by contacting only $\log N$ processors. A tree navigation can be done within $\log N$ steps regardless of the shape of the tree. We extend Skip Graphs/Nets [1, 11] to achieve such an efficient lookup.

Conceptually, a processor in a Skip Graph maintains $\log N$ levels of peer pointers, pointing to exponentially farther peers in the linear ordering of N processors. Figure 3 (b) depicts the overlay structure of the Skip Graph among the eight processors. Each processor uses a random membership vector to decide its peers. At level i, the peers are the nearest processors on the left and right sides with membership vectors that match the processor's membership vectors for the first i bits.

Brushwood routing depends on a linear ordering of partitions. In this sense, any linear space DHT routing facility can be used. We choose Skip Graphs for two reasons. First of all, Skip Graphs do not impose constraints on the nature and structure of keys. It can work with complex keys, like the variable-length Tree IDs, as long as there is a total ordering. Second, even if one can encode tree nodes into key values, such unhashed and often skewed keys can cause routing imbalances in some DHTs, as they use key values to decide the peering relation. Skip Graphs do not suffer from this problem because its peering is decided by purely random membership vectors, even though the keys are unhashed.



(a) Max Node Degree (b) Skew of Routing Load

Fig. 5. Routing Imbalance under Skewed Key Distribution

We simulated Chord [18] and Skip Graphs with a skewed key distribution to show the imbalance in routing. Figure 5 (a) depicts the maximal processor degrees of Chord and Skip Graphs with 1K$\sim$32K processors. The processor keys are derived from a normal distribution with standard deviation 0.125 in the range $[0, 1]$. With such unhashed keys, Chord processors falling into the sparsely populated regions will manage larger portions of the keyspace, and are therefore likely to have a large number of in-bound peers. Furthermore, the imbalance in peer distribution also leads to imbalance in routing costs. We route 1000 messages between random pairs of nodes. Figure 5 (b) shows the imbalance as the ratio of maximal routing load to mean load. We observed similar routing imbalances in Meghdoot, which employs CAN for routing in (skewed) subscription content space. We present this result in Section 6.

5 Maintaining the Partition Tree

In this section, we discuss the maintenance of the dynamic matching tree in a peer-to-peer setting. The major challenges are: 1) the frequent processor joins and departures, typically referred to as churn, and 2) balancing the workload among the dynamic processor set. Our design leverages Skip Graphs to achieve efficient routing while maintaining only a

logarithmic number of peers. Therefore, the processor joins and departures only result in small maintenance overheads. Balancing the workload associated with publish/subscribe events is important for the scalability of the system. The challenges that it presents in the context of the distributed matching tree differ from what previous work in DHTs have addressed. Therefore, we focus on this issue in this section. Our solution is based on a limited, loosely consistent knowledge about global load distribution. What is interesting about our scheme is that we use the distributed matching tree to aggregate this information.

5.1 Gossip-based Aggregation

In most peer-to-peer systems, periodical polling of peer nodes is necessary for detecting failures. We piggyback load information in the pair-wise heart-beat traffic between peers. Peer processors aggregate the global load information from these gossip messages. This approach is inspired by previous work [21].

Each processor maintains load summaries for the nodes in its partial tree view. This summary corresponds to the workload of the matching subtree rooted at the node and the resources available on the processors that maintain the subtree. In particular, it includes the following information: 1) the total number of subscriptions in the subtree; 2) the total rate of events visiting the subtree; 3) the total capacity of processors managing the subtree. The first two items show the load associated with subscription storage and event matching. The third summarizes the resource devoted for managing the load. We define capacity as the network bandwidth of the processor instead of storage, since this is the limiting factor for matching and delivering events. This information reflects the heterogeneity of participating processors. The load-to-capacity ratio in the summary indicates whether the subtree is overloaded or underloaded.

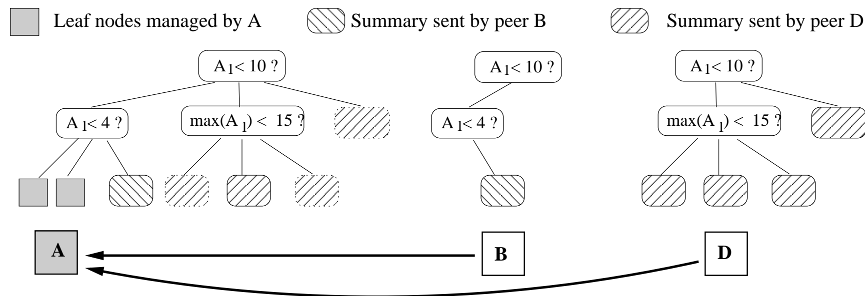


Fig. 6. Gossiping and aggregation of load information

Periodically, a processor sends to peers its load summaries about nodes along its Tree ID path (Section 4.1). Recall that this path stretches from the root to the first node (under pre-order) belonging to the processor. Figure 6 illustrates the Tree ID paths of peer B and D, and the gossip messages they send to A.

A maintains the storage and event processing load for the subtree it manages locally. After receiving load summaries from its peers, A can aggregate the load for the internal nodes in its partial tree. The summary about the root node gives the global load information. This information is loosely consistent. It is easy to see that the aggregation converges within $O(\log N)$ steps in a N-processor system, because information about one processor reaches all other processors within $O(\log N)$ forwarding steps, the diameter of a Skip Graph. With a

typical heart-beat interval of 30 seconds, the aggregation converges within several minutes, during which time the overall load is unlikely to change by a substantial amount.

5.2 Processor Join

When a new processor joins the system, it contacts a known processor P that is currently in the system. P uses the load summary in its partial tree view to direct the join request. It navigates the tree, locally, to find a subtree with a high load level, as determined by the ratio of total load to capacity associated with the subtree. If this subtree is remote or obscure (defined in Section 4.1), the join request is forwarded towards that subtree, and eventually reaches a peer Q with high load level. This forwarding process is similar to the distributed tree navigation for inserting subscriptions and matching events.

After receiving the join request, Q divides the set of leaf nodes it manages and hands over one half to the joining processor. If there is only one leaf node, or if one leaf node has significantly higher load than others, this leaf is partitioned using algorithms described in Section 3.5. The joining processor receives from Q the leaf nodes, which also determines the new Tree ID of the joining processor. The processor then joins the Skip Graph and establishes its partial tree view by contacting the peers.

Section 3.5 describes two strategies of leaf node partitioning: split or replicate. If the high load is caused by larger than average number of subscriptions, we choose one of the various options to partition the set of subscriptions among the new branches. If the load is caused by high event rate to the subscriptions, we may replicate the subscriptions in the new branches to spread out the event processing load.

5.3 Processor Departure and Failures

Processors in the system may leave gracefully or fail/quit silently without warning. In the former case, it notifies its peers of the intention to leave and hands over the set of leaf nodes and subscriptions to its left-hand side peer, and the Skip Graph will route corresponding messages to this peer after the processor's departure.

Failures and non-cooperative departures are detected by periodic heart-beat messages. If a processor P does not hear from a peer for several consecutive heart-beat intervals, this peer is marked as failed and is excluded from the partial tree view. If the peer is the immediate right-hand side peer, P takes over the responsibility of managing the leaf nodes of the failed peer. In order to avoid data loss, we can replicate subscriptions to left hand side peers during normal operation. This replication strategy is used in many peer-to-peer systems [16, 18, 15].

5.4 Reactive Load Balancing

Besides the load-balanced join process, reactive load balancing of heavily loaded processors is also desirable. Such imbalance may be caused by insertion of new subscription, transfer of data after peer departure, or change of event traffic pattern. Processors in the system detect load imbalance from the global load information. If a processor sustains significantly higher load than global average, it can start a load balancing process by navigating the distributed tree to find an underloaded processor. This processor is forced to quit its current position, offload its work to its neighboring processor, and rejoin the system as the overloaded processors' neighbor in order to take over half of the load from the overloaded processor.

6 Experimental Results

In this section, we present our experimental results. We use two very different real world datasets for publish/subscribe workload. We also evaluate system scalability with larger synthetic workloads. We start by describing the example applications and the datasets before presenting the experimental results.

6.1 Example Applications

Stock quote alert is a popular publish/subscribe service. Users subscribe to events about stock price changes and transaction volume fluctuations. Such services are usually implemented with DBMS triggers in a centralized server. Similar subscriptions that specify numerical data ranges may be found in other systems like monitoring and sensor networks. Therefore, we use stock quote alert as one of our representative applications.

We use the stock quote dataset collected by Gupta *et al.* to evaluate Meghdoot [9]. It was obtained from Yahoo! Finance [23] by downloading the daily quotes of 100 stocks from 2/Jan/1998 to 31/Dec/2002. This event set contains 115,353 events. The schema and value range of the events are summarized in Table 2. The data distribution is highly skewed. Most stock prices/volumes are within a relatively narrow range, except for a few high price/volume stocks quotes.

Attribute	*Date*	*Symbol*	*Open*	*High*	*Low*	*Close*	*Volume*
Type	String	String	Float	Float	Float	Float	Integer
Minimal	2/Jan/98	aaa	0	0	0	0	0
Maximal	31/Dec/02	zzzzz	500	500	500	500	310000000

Table 2. Schema of Stock Quote Events

Subscription	Prob.	Description
$\{Symbol = P_1 \wedge P_2 \leq Open \leq P_3\}$	20%	Notify when stock P_1 opens with price between P_2 and P_3.
$\{Symbol = P_1 \wedge Low \leq P_2\}$	35%	Notify when the price of stock P_1 is at most P_2.
$\{Symbol = P_1 \wedge High \geq P_2\}$	35%	Notify when the price of stock P_1 is at least P_2.
$\{Symbol = P_1 \wedge Volume \geq P_2\}$	5%	Notify when stock P_1 is traded at least P_2.
$\{Volume \geq P_1\}$	5%	Notify when any stock is traded more than P_1.

Table 3. Templates of Stock Quote Subscriptions

We follow the method used in [9] to generate stock subscriptions. Subscriptions randomly select one of five templates designed to model common user interests in stock events. Table 3 lists the subscription templates and their probabilities. The parameters are generated using random draws from uniform distributions over the data ranges of the corresponding fields, while maintaining the constraints. The fifth template is a "rare" case of a broad subscription that matches any stock with trading volume above a given parameter. In the real world, users are usually interested in events specific to a narrow group of stocks. Therefore, this template is assigned a relatively low probability.

While stock quote events exhibit a well-formed schema with numerical attributes, a number of applications use semi-structured data representations. We use the CiteSeer scientific literature digital library [6] as a representative data source for such applications. CiteSeer uses the Open Archives Initiative [14] protocol to publish the metadata of its literature collection. This metadata is encoded in XML, which accomodates semi-structured data and allows for efficient data manipulation. We parse the XML records published by CiteSeer to generate events one per publication, with the following extracted attributes:

Date, Title, Authors, Subject, and *References*. We further extract *Keywords* from the subject line by removing stop words and obtaining the stems of the remaining words. The *Authors, Keywords*, and *References* fields are represented with the Set type defined in Section 3.1. Note that some fields, like *References*, might be missing in some cases due to incomplete records. A total of 574,128 events are extracted.

We generate three types of subscriptions for our experiments:

- $\{Authors \ni P\}$: notify when the author list of a newly published paper includes P. We select parameter P from the list of authors appearing in the data set, with probability proportional to the occurrence frequency.
- $\{Keywords \supseteq P\}$: notify when a newly published paper includes the keyword list P. P is a set of one to three keywords selected randomly from the set of keywords in the data set, with probability proportional to keyword occurrence frequencies.
- $\{References \ni P\}$: notify when a newly published paper cites another document P. Again, P is randomly chosen according to data distribution.

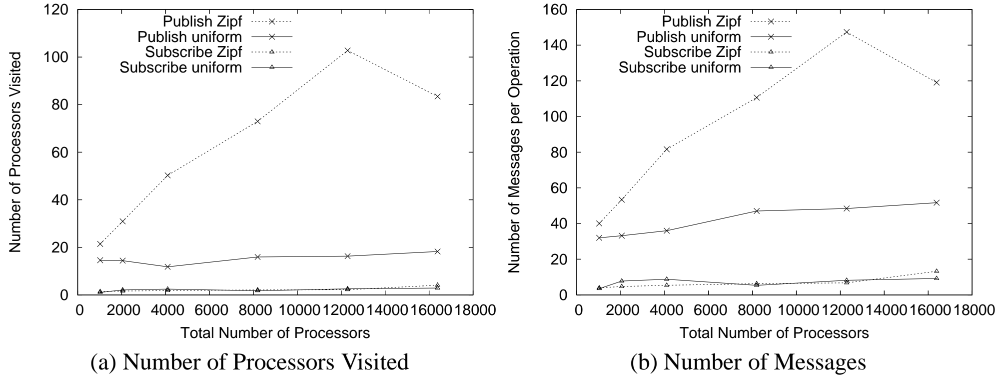
Besides the above two publish/subscribe data sets, we also use a synthetic workload to test system scalability, similar to that used in [4]. This workload uses events and subscriptions that specify one of more of 1000 numerical attributes. This synthetic workload models a general purpose publish/subscribe system that does not limit the users to a small set of pre-defined schemas. Each subscription specifies 1 to 10 predicates. Each predicate randomly selects an attribute, a comparison operator of $=$, $>$, $<$, $\leq$ or $\geq$, and a value between 0 to 999. We use either an uniform or a zipf distribution ($\alpha = 0.8$) to select the attributes. The operator and value fields are chosen uniformly randomly. Published events randomly specify between 1 to 20 attributes and their values, under the same distribution as for subscriptions.

We compare Brushwood matching tree against Meghdoot for the stock quote alert experiments. Meghdoot uses CAN to partition the multi-dimensional content-space to peer nodes. Meghdoot does not support the CiteSeer data set (due to the presence of set predicates) or the synthetic workload (due to the large number of attributes and the flexible event schema). So for these datasets, the experiments only evaluate our system under different parameters.

6.2 System Scalability

We first use the synthetic workload to evaluate system scalability. We simulate from 1024 to 16384 peer processors. The number of subscriptions is fixed at 1 million. The number of event messages is 110000. We start with a single processor and add the remaining at random intervals, in order to simulate a peer join process. In the mean time, we insert the subscriptions into the system. We count the number of messages forwarded for inserting subscriptions and publishing events as a measure of the communication cost. Some of the messages require further processing at the recipients: to insert a subscription or to match an event to local subscriptions. We measure this cost as the number of processors processing the request. We refer to this number as the textitspan of the operation, and the processors as *visited* by the operation. For subscriptions, it is the number of sites the subscription is replicated to. For events, it is the number of nodes that need to perform predicate evaluation or matching.

Figure 7 depicts the average number of processors visited and the average number of messages forwarded for a subscription/event. Even with 16384 nodes, a typical publishing



(a) Number of Processors Visited (b) Number of Messages

Fig. 7. Synthetic workload: cost vs. system scale

event spans less than 1% of the processors, showing good scalability. The maximal span we observed is about 250.

When attributes are selected using the Zipf distribution, the span of publishing events increases much faster than under uniform distribution. The reason is that a skewed distribution generates many similar subscriptions and events. In order to balance the load, these closely related subscriptions are partitioned across different processors. Events matching such subscriptions have to visit more partitions.

An interesting trend in Figure 7 is that the event span decreases when the number of processors increase from 12288 to 16384 (for Zipf distributed attributes). Meanwhile, the degree of subscription replication (indicated by the number of processors visited for subscription insertion) increases from 2 to 4. This is because that as more processors join, while the total number of subscriptions remains the same, our tree partitioning algorithm devotes the newly joined processors to store replicated subscriptions, thereby decreasing the number of processors that an event has to visit.

6.3 Stock Quote Alert

Next we evaluate the performance of our system and Meghdoot using the stock quote dataset. We scale the system from 128 processors to 8192 (the N parameters in the graphs). We also scale the number of subscriptions proportionally to the number of processors (100N).

Figure 8 shows the number of messages forwarded by subscription insertion and event matching as we increase the number of peer processors. Compared to Meghdoot, our scheme shows a substantially lower cost for processing events. This is first because we partition the subscription set based on data distribution. Meghdoot uses CAN's partitioning method that splits a zone into halves of equal sizes (The reason for this regular split is to avoid interleaving of the zone spaces that can significantly increase the number of peering zones.) Therefore it suffers load imbalance under the highly skewed dataset. In order to alleviate this imbalance, Meghdoot replicates the overloaded nodes, resulting in a higher number of subscription messages. Another reason is the flexible value partitioning method used in the matching tree (Section 3.4). Meghdoot partitions the subscriptions by Min/Max range specified for the attributes. This approach splits the subscriptions into non-overlapping sets, but an event may need to visit both zones after the split. We use value

range partitioning method that allows events to visit only one branch after the partition. Our approach also replicate some subscriptions, but only limited to broad ones. So the subscription cost is still lower than that of Meghdoot.

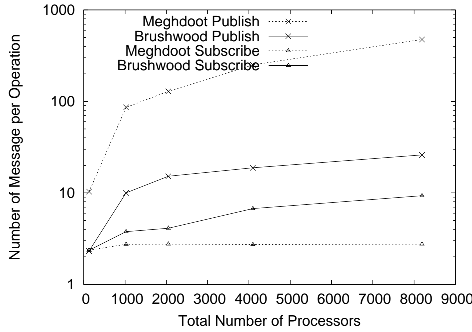


Fig. 8. Stock: Number of Messages vs. System Scale

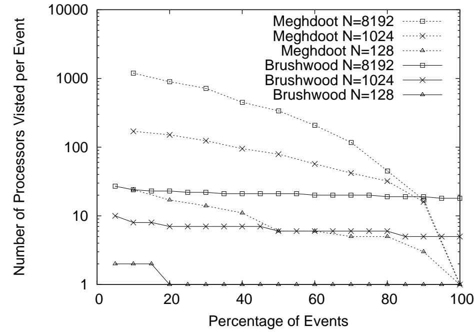
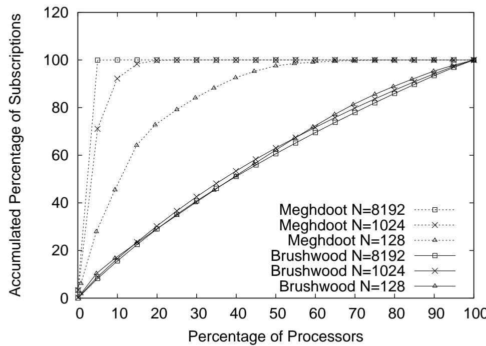


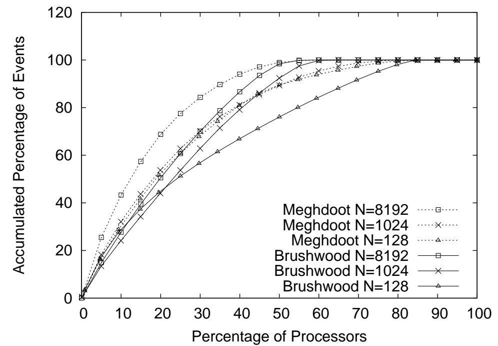
Fig. 9. Stock: Distribution of Number of Processors Visited for Publishing an Event

Figure 9 shows the histogram of event spans (the number of processors visited by the event). Under all three settings of system scale, our scheme demonstrates relatively small and stable span, due to reasons discussed above.

Next, we compare the load balance of the two systems. We consider several aspects of load balance: subscription storage, event matching, and routing state. Routing state is represented by the number of peers that processors maintain.



(a) Subscription Storage



(b) Event Processing

Fig. 10. Stock: load distribution

Figure 10 (a) presents the cumulative distribution (CDF) of the number of subscriptions managed by the processors. Our system exhibits evenly balanced storage loads, while most of the subscriptions in Meghdoot are managed by a small number of nodes. The imbalance in Meghdoot is due to the fact that only some of the zones (the portion of the CAN space above the diagonal plane) are used to store subscriptions. Moreover, the constraint of equal-space partitioning also limits its ability to achieve balanced load under skewed data distribution.

Figure 10 (b) depicts the CDF of the percentage of events prcessed by the processors. Note that each event may be examined by multiple processors, so the total is higher than the number of events submitted to the system. Our system shows better load balance in event

processing, because the subscriptions are more evenly partitioned among the peers. Some of the subscriptions match very broad range of events (like those only specifying *Volume* in Table 3), Both Brushwood and Meghdoot replicate some subscriptions to share the event matching load. Therefore, there is not a significant difference between the two schemes in balancing the loads associated with event processing.

We discussed the routing state balance problem in Section 4.2. In Skip Graphs, the peering relationship is decided by random membership vectors, and hence is not affected by skewed key distributions. Meghdoot uses CAN for overlay routing, which decides peering by zone neighborhood. Therefore, larger zones may have more peers if the zones are partitioned into different sizes under a skewed data distribution. In a high dimensional space, this imbalance is more significant since zones can make contact along more dimensions. Figure 11 confirms this intuition.

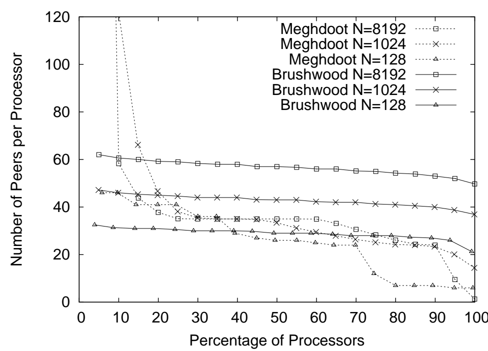
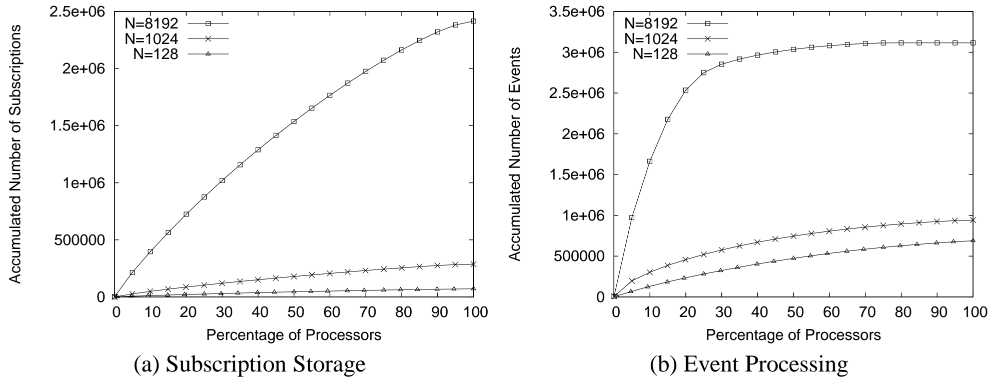


Fig. 11. Stock: Distribution of Number of Peers per Node

6.4 Literature Reference Notification

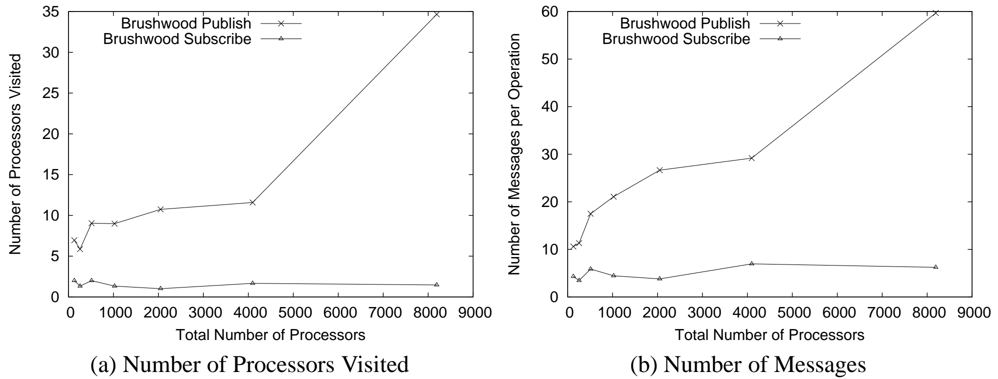
Now we present the results of the CiteSeer experiments. We use simulation settings similar to the above tests, except that the subscriptions choose parameter values based on a real distribution derived from the data set, instead of using uniform random distributions. Figure 12 shows the CDF of the subscription storage and event matching load on the processors. Although the contents of subscriptions and events have skewed distributions, the load balancing mechanisms in Brushwood ensure good load balance.

Figure 13 (a) (b) shows the cost of inserting subscriptions and the cost of processing events. Both the number of messages and the number of nodes visited are small. Since the attributes *Authors*, *Keywords*, and *References* are of *Set* type, the span of subscription and event messages is mainly decided by the number of items specified. In this real-world data set, the number of authors, keywords and references are usually small. Therefore the Brushwood approach performs well. However, we do observe a sharp increase in publishing cost as the number of processors is increased from 4096 to 8192. This is due to the dynamic load balancing mechanism discussed in Section 5.4. As the peer population increases, popular subscriptions can receive a significant number of subscribers. Therefore, peers maintaining them get overloaded and split their load to more processors. As a result, events involving such subscriptions have to flood more peers, while each peer still maintains a reasonable share of load (Figure 13). We did not observe such a trend in previous experiments because their subscription values are drawn from a uniform distribution. Though there is an in-



(a) Subscription Storage (b) Event Processing

Fig. 12. CiteSeer: cumulative load distribution



(a) Number of Processors Visited (b) Number of Messages

Fig. 13. CiteSeer: cost of operations vs. system scale

crease in publishing cost, we do note that the reactive load balancing mechanism manages to balance load even in the face of skewed subscription patterns.

7 Conclusions

In this paper, we propose a content-based publish/subscribe middleware built by distributing a matching tree over a peer-to-peer system. The main contribution is in the decentralized navigation and management algorithms for the distributed matching tree in peer-to-peer settings. Our system achieves efficient event matching while requiring only small amounts of state to be maintained by the peers. Processors in the system build partial views of the global tree based on information about only a logarithmic number of peers. Therefore, the system provides high scalability. Compared to other peer-to-peer approaches, it imposes no restrictions over the schemas associated with subscriptions and events. The use of a matching tree provides more generality and extensibility in the types of data and predicates that can be supported. The peer-to-peer tree also provides aggregated load information that assists reactive load balancing. Experiments demonstrate that the proposed design effectively supports real world subscription scenarios. Besides publish/subscribe, we have used the Brushwood framework to build other applications, including high dimensional index and distributed file systems. We believe that the combination of techniques brought to-

gether in Brushwood (such as the ability to support search tree data structures, efficient decentralized navigation using partially consistent views, load-balance using aggregated information) shows promise as a powerful toolkit for building scalable distributed applications.

References

1. J. Aspnes and G. Shah. Skip Graphs. In *Proceedings of Symposium on Discrete Algorithms*, 2003.
2. G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *ICDCS*, 1999.
3. S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable application layer multicast. In *SIGCOMM*, 2002.
4. A. Carzaniga and A. L. Wolf. Forwarding in a content-based network. In *SIGCOMM*, 2003.
5. M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. SCRIBE : A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications*, 2002.
6. CiteSeer. http://www.citeseer.org/.
7. F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *SIGMOD*, 2001.
8. J. Gough and G. Smith. Efficient recognition of events in a distributed system. In *Proc. of the 18th Australasian Computer Science Conference*, 1995.
9. A. Gupta, O. D. Sahin, D. Agrawal, and A. E. Abbadi. Meghdoot: content-based publish/subscribe over p2p networks. In *Proc. of the 5th ACM/IFIP/USENIX International Conference on Middleware*, 2004.
10. E. N. Hanson, M. Chaabouni, C.-H. Kim, and Y.-W. Wang. A predicate matching algorithm for database rule systems. In *SIGMOD*, 1990.
11. N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. In *USITS*, 2003.
12. Y. hua Chu, S. G. Rao, and H. Zhang. A case for end system multicast. In *SIGMETRICS*, 2000.
13. Object Management Group. Corba event service specification (version 1.1), March 2001.
14. Open Archives Initiative. http://www.openarchives.org/.
15. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proceedings of ACM SIGCOMM*, 2001.
16. A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Middleware*, 2001.
17. B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceeding of AUUG97*, 1997.
18. I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, 2001.
19. D. Tam, R. Azimi, and H.-A. jacobsen. Building content-based publish/subscribe systems with distributed hash tables. In *Internation Workshop on Databases, Information Systems and Peer-to-Peer Computing*, 2003.
20. TIBCO. http://www.tibco.com/.
21. R. van Renesse and K. P. Birman. Scalable management and data mining using astrolabe. In *IPTPS*, 2002.
22. Y.-M. Wang, L. Qiu, D. Achlioptas, G. Das, P. Larson, and H. J. Wang. Subscription partitioning and routing in content-based publish/subscribe networks. In *16th International Symposium on Distributed Computing*, 2002.
23. Yahoo! Finance. http://finance.yahoo.com/.
24. C. Zhang, A. Krishnamurthy, and R. Y. Wang. Brushwood: Distributed trees in peer-to-peer systems. In *IPTPS*, 2005.