Peer-to-Peer Netw Appl (2008) 1:3–17
DOI 10.1007/s12083-007-0001-3

An efficient and scalable framework for content-based publish/subscribe systems

Yingwu Zhu · Haiying Shen

Received: 21 March 2007 / Accepted: 22 November 2007 / Published online: 10 January 2008

Abstract Challenges for content-based publish/subscribe systems include efficient subscription management and event matching, load balancing, and efficient and scalable event delivery. This paper presents an efficient and scalable framework for content-based publish/subscribe systems. We propose using K-D trees to dynamically partition and organize subscriptions, thereby preserving subscription locality, minimizing event matching load and ensuring load balance across nodes. We propose an efficient event delivery mechanism that cleverly exploits embedded trees in distributed hash tables to disseminate events. We show that the latency of event publication and delivery is low. The event delivery mechanism can deliver events to a large number of subscribers at low latency and overhead, consuming modest bandwidth.

Keywords Distributed hash tables · Content-based publish/subscribe · Subscription management · K-D trees · Event matching · Event delivery

Y. Zhu (✉)
Department of CSSE, Seattle University,
Seattle, WA 98122, USA
e-mail: zhuy@seattleu.edu

H. Shen
Computer Science and Computer Engineering Department,
University of Arkansas, Fayetteville
AR 72701, USA
e-mail: hshen@uark.edu

1 Introduction

Content-based publish/subscribe (pub/sub) [1] is a powerful paradigm for information dissemination from publishers (data/event producers) to subscribers (data/event consumers) in large-scale distributed networks. A data event specifies values of a set of attributes associated with the event. Subscribers register their interests in future events through subscriptions, which can be very expressive, and specify complex filtering criteria by using a set of predicates over event attributes. Upon receiving an event published by a publisher, the system matches the event to the subscriptions which serve as filters and delivers the event to the matched subscribers. A content-based pub/sub system is required to store the subscriptions installed by the users and upon an event, find all subscriptions matching the event and deliver the event to the matched subscribers.

Content-based pub/sub is valuable to many applications, including personalized information dissemination (e.g., online stock quotes), monitoring and alerting (e.g., sensor networks), and application integration. Current content-based pub/sub systems are either centralized or distributed. Centralized solutions [2], while simple, have an inherent scalability problem. Distributed systems [3–6], which usually use multicast trees to deliver events, however, may suffer from the following limitations. First, most systems distribute subscriptions *randomly* across nodes and fail to preserve subscription locality, thereby increasing system-wide event matching load. Second, using dedicated multicast trees for event dissemination incurs nontrivial bandwidth cost in per-tree construction and maintenance, especially in dynamic systems where nodes join or leave at will. Third, event delivery is inefficient in terms of

bandwidth. Finally, with few exceptions [7], most solutions have a load balance issue as subscriptions and events in real-world applications are highly non-uniform.

Distributed hash tables (DHTs) [8–11] are particularly attractive for supporting content-based pub/sub systems due to their scalability, decentralization, fault-tolerance and self-organization. In this paper, we present a novel ap**PR**roach to **E**fficient and **S**calable content-based publish/**S**ubscribe (pub/sub) systems (**PRESS**) on top of DHTs. In particular, the goals of this proposed approach that make our contributions, are to meet the following requirements.

- **Subscription locality preservation.** subscription management is based on content such that similar subscriptions are stored close together on a (or a small number of) node(s) while dissimilar subscriptions will be distributed across different nodes. Consequently, the system-wide event matching load can be minimized, by only matching those subscriptions most likely relevant to the event. In addition, with subscription locality, current techniques of collapsing similar subscriptions [12] and subscription covering and merging [13], as a means of reducing the matching load which is in proportion to the number of subscriptions, will be more effective.
- **Load-balancing.** Real-world subscriptions can be highly non-uniform. Locality-aware subscription distribution can result in skewed load distribution among nodes. The system thus should balance the load such that each node that stores subscriptions should be responsible for roughly same number of subscriptions.
- **Light-weight, efficient and timely event delivery.** Current solutions use multicast trees for event dissemination, introducing nontrivial cost of per-tree construction and maintenance. Hence, the system should minimize or even eliminate such cost imposed on the underlying DHT. Moreover, the event delivery should be efficient in terms of bandwidth cost and timely in terms of user-perceived latency.

The remainder of the paper is structured as follows. Section 2 gives an overview of related work. Section 3 presents the framework of PRESS. We discuss subscription organization in Section 4, describe event publication and matching in Section 5, and present event delivery in Section 6. Section 7 discusses experimental setup and results. We conclude the paper in Section 8.

2 Related work

2.1 Content-based pub/sub model

Pub/sub systems can be classified into two categories: *topic-based* and *content-based*. Topic-based pub/sub systems assign each event to one of a set of predefined topics (also referred to as groups, channels or subjects). Each event itself specifies the topic that is associated with the event. A user subscribes to a set of topics he/she is interested in and is informed of all the events which are associated with these topics. Topic-based pub/sub systems take only coarse-grained subscriptions. As a result, a user has to receive all events pertinent to a topic though he/she might be interested in only a subset of the events. By contrast, content-based pub/sub systems allow fine-grained subscriptions by enabling restrictions on the event content. A subscriber can specify multiple predicates as a subscription and only those events satisfying all the predicates are notified to the subscriber. Subscriptions in content-based pub/sub systems are more expressive, which makes the system harder to implement.

Fabret et al. [1] proposed a content-based pub/sub scheme with multiple attributes, defined as: $S = \{A_1, A_2, \cdots, A_n\}$, where each A_i corresponds to an attribute. Each attribute has a *name*, *type*, and *domain*, and can be specified by a tuple (*name*, *type*, *min*, *max*). The attributes are identified by their unique names. *type* could be *integer*, *float*, and *string*, etc. The *min* and *max* define the range of domain values taken by the given attribute. An event is a set of $<$ *attribute*, *value* $>$ pairs, and it can be represented as $e = \{A_1 = c_1, A_2 = c_2, \cdots, A_n = c_n\}$. A subscription is a conjunction of predicates over one or more attributes. Each predicate has a *name*, *type*, *operator* and *value* and is used to specify a constant value or range for an attribute. An example subscription is $(A_1 = v1) \wedge (v_2 \leq A_3 \leq v_3)$. An event e matches a subscription s if each predicate of s is satisfied by the value of the corresponding attribute contained in e. The content-based pub/sub system stores the subscriptions from all subscribers and given an event, finds all subscriptions matching the event and delivers the event to the subscribers. In this paper, we base our discussion on this model.

2.2 Centralized pub/sub systems

One example system is Elvin [2]. It uses a central server that stores all the subscriptions, evaluates the subscriptions upon events and delivers events to the matched subscribers. Centralized solutions, however, have an inherent scalability problem as the number

of events and subscriptions in the system increases. Hence, Fabret et al. [1] proposed novel data structures and application-specific caching policies and query processing to support high rates of subscriptions and events in the system. Specifically, Fabret et al. used the data structures including a set of indexes, a predicate bit vector and a cluster vector to achieve efficient event matching that is based on clustering and maximizes temporal and spatial locality. However, restrictions have to be placed on subscriptions such that they must contain at least one equality predicate, sacrificing flexibility and expressiveness of subscriptions.

2.3 Decentralized pub/sub systems

Many distributed pub/sub systems [3–6, 14–16] have been proposed by using routing trees to perform event delivery based on multicast techniques. Siena [15] builds a symmetric spanning tree and each pub/sub server can be a publisher or subscriber. Gryphon [16] organizes the pub/sub network into a single-source tree and proposes a link matching algorithm to forward events towards directions of matching subscriptions.

In MEDYM [6], some matcher nodes matches an event to the subscriptions and obtains a destination list of the matched subscribers. Then, the event delivery message containing the destination list is routed through a dynamically generated dissemination tree with the help of topology knowledge. Our event delivery mechanism is similar to MEDYM in that the event message carries a subscriber list. However, our proposed approach differs from MEDYM in that it exploits embedded trees in the underlying DHT to deliver events, thereby incurring no cost in multicast tree construction and maintenance.

To improve event routing efficiency, Kyra [17] proposes content clustering to create multiple pub/sub networks each of which is responsible for a subset of the content space. In the similar vein, HYPER [18] dynamically identifies a number of virtual groups based on common subscriptions. The event message is matched only at the entry point of a group and then forwarded to the group members if the group is a match.

2.4 DHT-based pub/sub systems

We have seen many attempts in designing DHT-based pub/sub systems. Scribe [19] and Bayeux [20] are essentially topic-based pub/sub systems. They do not directly support content-based pub/sub services. SplitStream [21] is an application-level multicast system built from Scribe for high-bandwidth data dissemination. To balance forwarding load over participating nodes with heterogeneous bandwidth constraints, SplitStream splits content into k stripes each of which corresponds to a Scribe multicast tree. Tam et al. [22] proposed a content-based pub/ sub system built from Scribe. The system places some restrictions on subscriptions and thus sacrifices expressiveness in subscriptions.

Terpstra et al. [23] proposed a content-based pub/sub system built on top of Chord. Both filter updates (e.g., due to subscribing and un-subscribing) and event routing actually use a broadcasting algorithm. Triantafillou et al. [24] proposed to distribute subscriptions on the Chord nodes based on the keys produced by hashing the attribute and its values. If the subscription specifies a range over an attribute, the subscription would be stored on a number of nodes by hashing the attribute and each of its possible values within this range. The main drawback is that subscription installation and update are expensive due to the large number of nodes and messages potentially involved.

Reach [25] maintains a semantic overlay network and uses a Hamming-distance based routing scheme. Each node serves as a rendezvous point for those subscriptions with suffix matching the node's identifier. In the similar vein, HOMED [26] maintains a semantic overlay where each node's identifier is derived from its subscriptions. However, they have two main limitations. First, they assume a globally-static attribute space. Second, they have a load balancing issue since non-uniformly distributed subscriptions would cause unevenly distributed nodes on the overlay. Meghdoot [7] is based on CAN. Subscriptions are stored on a zone according to the coordinate determined by event attribute values. Considering skewed distributions of subscriptions and events in a real application, Meghdoot addresses the load balancing issue by zone splitting and zone replication. The major limitation of Meghdoot is that the overlay dimension is proportional to the number of event attributes.

Our previous work Ferry [27] provides a preliminary study of exploiting the embedded trees in DHTs to deliver events. The work presented in this paper is motivated by the lessons learned from Ferry. It distinguishes itself from Ferry by proposing a new architecture that aims to preserve subscription locality in subscription management, minimize event matching load, balance load across nodes, and offer efficient and scalable event delivery.

2.5 Other related work

K-D tree [28] is a widely used index tree for high dimensional data. We use K-D tree to distribute subscriptions, thereby preserving subscription locality and reducing

event matching load system wide. Brushwood [29] is another example to use K-D tree to support locality-sensitive applications in the P2P environments, by organizing the K-D tree leaf nodes into skip graph for complex queries (e.g., range queries). Inspired by Prefix Hash Tree (PHT) [30], we layer the K-D tree on top of a DHT to support subscription distribution and event publication.

One challenge faced by content-based pub/sub systems is the ability to handle a vast amount of subscriptions and events. Li et al. [13] proposed using modified binary decision diagrams to represent subscriptions by exploiting subscription covering and merging. Aguilera et al. [12] proposed sublinear matching algorithms based on building subscription trees that collapse similar subscriptions. The proposed subscription covering and merging techniques complement our work in reducing subscription management and event matching load.

3 Framework

The framework of PRESS is based on the following three key mechanisms:

Subscription Organization Mechanism (SOM). SOM uses K-D tree techniques to organize subscriptions in a hierarchical tree manner, and stores the subscriptions only on leaf nodes. The goals of SOM are to meet the following requirements. First, it aims to preserve locality of subscriptions, i.e., similar/relevant subscriptions are stored on a (or a small number of adjacent) leaf node(s). Second, each leaf node is responsible for roughly same number of subscriptions, ensuring load balance across leaf nodes. SOM layers the tree structure on top of a DHT, by which each tree node is hosted by a DHT node and the tree inherits fault-resilience and self-organizing properties of the underlying DHT. Subscription installation is a process of tree navigation from the tree root to the corresponding leaf node(s). The subscription installation, however, has two drawbacks: (1) It may involve multiple overlay hops since the tree spans the DHT overlay, thereby incurring high latency. (2) Every installation goes through the root, creating a potential bottleneck. Hence,PRESS uses *KDT-lookaside cache* at client/subscriber side to alleviate the problems.

Event Publication and Matching Mechanism (EPMM). EPMM allows event publishers to publish an event along the K-D tree to the leaf node that stores the subscriptions relevant to the event. The leaf node then matches the event to the subscriptions and starts delivering the event to the matched subscribers. Similar to subscription installation, event publication could incur high publication latency and create a potential bottleneck on the tree root node. To alleviate the problems, the KDT-lookaside cache is employed at the client/publisher side. Event matching algorithms can adopt current subscription covering and merging techniques [12, 13] at the leaf node to reduce subscription management and event matching load.

Event Delivery Mechanism (EDM). EDM is virtually maintenance-free. It smartly exploits embedded trees inherent in the underlying DHT to deliver events, thereby eliminating the cost of multicast-tree construction and maintenance. After a leaf node matches an event to the subscriptions stored on it, the leaf node multicasts the event through the corresponding DHT links of its DHT host node. The event is then disseminated along the embedded tree rooted at the DHT node hosting the leaf node, and finally reaches each subscriber. EDM aggregates messages along event dissemination paths, thus reducing the number of event delivery messages and bandwidth consumption. Moreover, exploiting DHT links for event delivery, EDM has three major advantages: (1) The underlying DHT maintenance messages could be piggybacked onto the event delivery messages to reduce the DHT maintenance cost. (2) Proximity neighbor selection (PNS) in the underlying DHT, as a means of improving routing performance, makes event dissemination along the embedded tree proximity-aware, achieving good event delivery performance. (3) The fault-tolerance and self-organizing nature of DHT overlays makes event delivery along the DHT links resilient to node/link failures.

For ease of exposition, the discussions of PRESS is based on the content-based pub/sub scheme described in Section 2.1.

4 Subscription organization

A subscription s in the system is expressed by a pair (sid, p), where sid is the subscriber's node ID[1] (*subscriber ID* for short) and p is a conjunction of predicates specifying the subscriber's interests (e.g., $(A_1 = c_1) \wedge (c_2 \leq A_2 \leq c_3)$).

4.1 Using K-D tree to organize subscriptions

K-D tree (KDT) [28] is a widely used index tree for high dimensional data. Using KDT, PRESS dynamically partitions the subscription space of a pub/sub

[1] In DHTs, node ID can be either obtained when a node joins the overlay or determined by hashing its IP address or public key.

scheme S [1] into smaller and smaller regions. A KDT here is essentially a binary trie in which each node corresponds to a subscription region (i.e., a region in the multi-dimensional space) and only leaf nodes store subscriptions. Each internal node specifies a partition attribute $attr_{split}$ and a split position pos_{split} along this attribute, and splits itself into two children. Each node has a distinct *label* which is derived recursively: Given a node with label L, its left child and right child nodes are labeled as $L0$ and $L1$ respectively. The root node has a label "0" by default. Each node of the KDT has either zero or two children.

Except the root, each node maintains a split history, a list of tuples $< attr_{split}, pos_{split}, 0/1 >$ (where 0/1 represents the path to the left/right child. The left child is responsible for the subscriptions with ($attr_{split} \leq pos_{split}$) and the right child is responsible for the subscriptions with ($attr_{split} > pos_{split}$)). The split history is also derived recursively. Consider a node with split history H, partition attribute $attr_{split}$ and split position pos_{split}. The split histories for its left child and right child are $H_L = H \cup \{< attr_{split}, pos_{split}, 0 >\}$ and $H_R = H \cup \{< attr_{split}, pos_{split}, 1 >\}$, respectively. Each node is responsible for a subscription space specified by its split history. The label of a node represents the path in the KDT from the root to the node and can be derived from its split history.[2] Figure 1 illustrates a KDT in which node E maintains a split history of $\{< A_1, c_1, 0 >, < A_2, c_2, 1 >\}$ and its label is "001". Node E stores the subscriptions with $(A_1 \leq c_1) \wedge (A_2 > c_2)$.

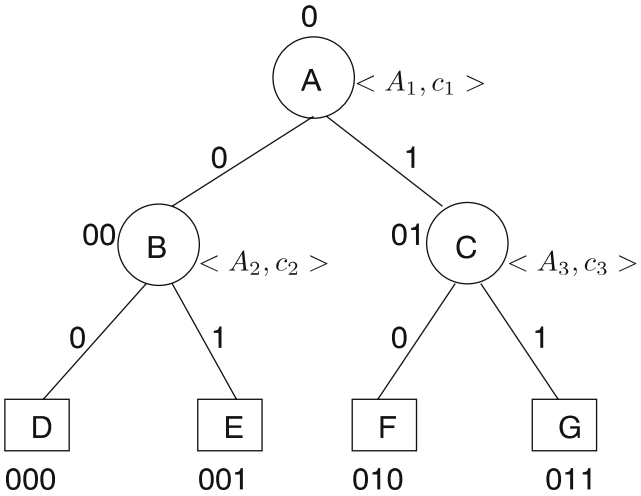


Fig. 1 Illustration of a KDT. $< A_i, c_i >$ is the partition attribute and split position

4.2 Subscription installation

Given a KDT described above, subscription installation is a process of tree traversal from the root until meeting a leaf node which is the right place to store subscriptions. To ensure load balance among leaf nodes, the KDT imposes a threshold T on the number of subscriptions each single leaf node can store. When a leaf node fills to T, it must split into two descendants by partitioning the most distinguishing attribute along the median of the attribute values in the subscriptions.[3] The leaf node then transfers the corresponding partition of the subscriptions and gives the corresponding copy of the split history to the two new child nodes, records the split attribute and position, and marks itself as an internal node.

Consider the KDT shown in Fig. 1. To install a subscription $s = (sid, (A_1 < c_1) \wedge (A_2 > c_2))$, the subscription is first forwarded to the root A which in turn forwards s to node B according to its split attribution and position $< A_1, c_1 >$. B performs similar operations and forwards s to the leaf node E, which determines whether to store s or trigger a split operation if the number of subscriptions reaches T.

However, subscription installation may cause split of subscription along the traversal path, resulting in a small number of subscriptions stored on a small number of different leaf nodes. Consider a subscription $s = (sid, (A_1 < c_1) \wedge (c_4 < A_2 < c_5))$, where $c_4 < c_2 < c_5$. The installation starts from the root A and then reaches node B. Upon s, node B splits s into two subscriptions $s_1 = (sid, (A_1 < c_1) \wedge (c_4 < A_2 \leq c_2))$ and $s_2 = (sid, A_1 < c_1) \wedge (c_2 < A_2 < c_5))$, according to the split attribute A_2 and position c_2. Then, $s1$ and $s2$ each takes different paths and finally reaches the leaf nodes D and E respectively. Node D and E each will independently determine whether to simply store the subscription or cause expansion of the KDT by a split operation. Note that $s1$ and $s2$ could be stored in a simpler form on D and E respectively, i.e., $s_1 = (sid, (A_1 < c_1) \wedge (c_4 < A_2))$ and $s_2 = (sid, (A_1 < c_1) \wedge (A_2 < c_5))$.

4.3 Unsubscribing

If we associate each subscription with a TTL (time-to-live), then subscribers do not need to perform unsubscribing operation. The main drawback is that

[2] We need to prefix "0" to the string of 0s and 1s extracted from the split history.

[3] This is two-fold. First, the subscriptions in the two partitions are probably less similar if we choose the most distinguishing attribute as the split attribute. Second, splitting along the median of the attribute values is to balance the load between two descendant nodes.

subscribers need to refresh their subscriptions before the subscriptions expire. If a subscription does not have TTL, then subscribers need to explicitly remove their subscriptions installed previously. The unsubscribing operation is essentially similar to the subscription installation. The main difference is that subscription removal may cause shrink of the KDT, i.e., it coalesces two sibling leaf nodes into a single parent node if the total number of subscriptions on the two leaf nodes drops below T. In this case, the parent node takes all the subscriptions stored on its two child nodes, trim the two child nodes, and becomes a leaf node. The merge operation is essentially reverse of splits during subscription installation and can be performed lazily in the background.

4.4 Layering KDT on top of a DHT

DHTs distribute objects in a load-balanced and deterministic manner and they allow efficient lookup by their IDs. Recall that KDT nodes have distinct labels. We layer a KDT on top of a DHT as follows. For each KDT node (say, A), we produce a unique ID by hashing its label L_A. Resorting to the *put*/*get*/*remove* interface offered by the underlying DHT, each KDT node A is hosted by a DHT node (represented as $DHT\_host(L_A)$) which is responsible for the generated ID $h(L_A)$ (in Chord, the DHT host node is the successor of the generated ID). In addition, unsubscribing may cause shrink of the KDT, which can also be completed by using *put*/*get*/*remove* interface. Leveraging DHT's self-organizing property and data replication mechanisms under node churn, the DHT-hosting KDT inherits all of the resilience and failure recovery properties of the underlying DHT. One distinction should be made between a KDT node and DHT node: the node on the KDT is called KDT node while the peer node on the DHT overlay is called DHT node; A DHT node hosts the KDT nodes whose IDs fall into the DHT node's responsible ID region.

Given a KDT (as shown in Fig. 1) layered atop the DHT, the installation of subscription $s = (sid, (A_1 < c_1) \wedge (A_2 > c_2))$ is performed as follows. The subscriber first routes s to the DHT node hosting the KDT node A with label "0", represented as DHT_host("0"). Upon receiving s, A uses its split attribute A_1 and position c_1 to make a decision and routes s to the DHT node hosting the KDT node B with label "00", DHT_host("00"). Similarly, B uses its split attribute A_2 and position c_2 to make a decision and finally routes s to the DHT node hosting the KDT leaf node E with label "001", DHT_host("001"). Upon receiving s, E determines whether to store s or trigger a split operation. It is worth pointing out the subscription routing is based on the underlying DHT's routing mechanism. The subscription installation takes $d \cdot \log N$ overlay hops, where d is the average depth of the KDT and N is the number of nodes in the underlying DHT.

4.5 Discussion

Using KDTs to organize subscriptions guarantees subscription locality such that subscriptions are similar/relevant on each leaf node and subscriptions close to each other in the subscription space are on adjacent leaf nodes. Moreover, the subscription number threshold T aims to balance the load among leaf nodes, ensuring that the number of subscriptions stored on a single leaf node is within $[T/2, T]$.

However, subscription installation described above has two main drawbacks. First, it needs to traverse the KDT spreading on top of a DHT, involving a number of overlay hops. The multiple overlay hops could be translated into high latency[4] since DHT nodes could be scattered in the Internet and a single DHT hop could be high-latency WAN link. Second, it creates potential bottleneck at the KDT root since each traversal starts from the root. Below, we discuss how to use KDT-lookaside cache at client/subscriber side to improve performance and avoid the potential bottleneck.

4.6 Caching to improve performance

The primary purpose of KDT-lookaside cache is to (1) improve performance and (2) bypass the root and start subscription installation from lower levels in the KDT. To achieve this, each subscriber maintains a KDT-lookaside cache that keeps track of the shape of the KDT based on previous subscription installation operations. Each cache entry consists of a KDT node's label and split history. Subscription installation is first checked against the lookaside cache. The cache returns the longest matching-prefix L_{pfx} of KDT node's label.[5] Formally speaking, the cache returns the smallest multi-dimensional region (corresponding to the pub/sub scheme S) that encloses the region specified by s. Then, the installation starts from the KDT node with label L_{prf} (or DHT node $DHT\_host(L_{prf})$).

[4] Subscription installation may not be so performance critical as event publication.

[5] Search for the longest matching-prefix for a subscription s in the cache is a greedy algorithm which starts checking from the entry with the longest label. Each cache entry uses its split history to find the longest prefix of its label for s and we choose the longest matching-prefix among all the entries.

(One optimization is that the client/subscriber can first split the subscription accordingly by referring to the cache, allowing the subscription installation operations to start from lower levels of the KDT. Split of subscriptions at the client-side requires the subscriber issue multiple installation request messages. We currently do not apply this optimization in our experiments.) When a subscription reaches a KDT leaf node, the leaf node informs the subscriber of its label and split history, allowing the subscriber to record the information in its cache. The newly added cache entry may evict those entries whose labels are prefix of the newly added label to make efficient use of cache space.

However, cache entries could be outdated due to the shrink of the KDT which has invalidated some KDT nodes. For example, if the KDT node with label L_{prf} returned by cache lookup are not present any more due to merge operations performed on the KDT, the subscriber can retry the installation operation from a KDT node with a shorter label, i.e., a prefix of L_{prf} by trimming the rightmost character(s). In the meantime, the subscriber invalidates the corresponding cache entry, and adds a new cache entry that has been verified by the last successful retry if it does not exist in cache yet.

5 Event publication and matching

5.1 Event publication

Event publication is essentially similar to subscription installation. Formally speaking, viewing e as a point in the multi-dimensional space corresponding to the pub/sub scheme S, event publication is a process of projecting the point into a multi-dimensional region that encloses this point and is maintained by some KDT leaf node. This leaf node should store the subscriptions that enclose this point in the multi-dimensional space. (If a subscription encloses the point, then the subscription is a match for e) As shown in Fig. 2, publisher P publishes an event e, which is first routed to a KDT node A. Upon receiving e, node A bases its decision on the split attribute and position and then routes e to node B. Similarly, node B makes its own decision and routes e to node C that is a leaf node enclosing e. Node C matches e to the subscriptions and finally starts delivering the event to the matched subscribers. It is worth pointing out that using KDT to organize subscriptions and perform event publication ensure that, events will be published to the leaf node that stores the most relevant subscriptions, thereby minimizing event matching load system wide. For example, many existing distributed pub/sub systems distribute subscriptions randomly across nodes. As a result, events have to be published to all these nodes and matched to the subscriptions scattered on all these nodes.

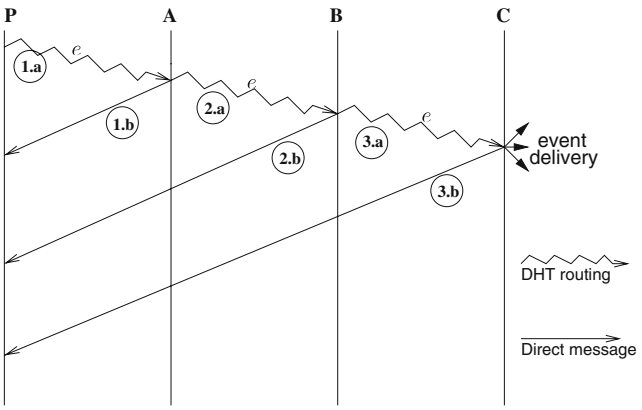


Fig. 2 Event publication. P is the publisher who publishes an event e. A, B and C are KDT nodes and C is a leaf node

Similar to subscription installation, event publication presented above suffers from two limitations. First, event publication takes $d \cdot \log N$ overlay hops, and thus may introduce high latency. Second, event publication requires KDT traversal starting from the root, making the root a potential bottleneck.

5.1.1 Caching to improve performance

To address the aforementioned problems, we use KDT-lookaside cache on the client/publisher side. Note that each node on the DHT overlay can be a publisher and subscriber. Hence, the lookaside cache keeps track of the shape of the KDT based on past subscription installation and event publication operations. Consider the event publication example illustrated in Fig. 2. The information (i.e., split history and node label) included in response messages 1.b, 2.b and 3.b are used to refresh old cache entries or add new cache entries. Note that the response messages could be sent back to the publisher asynchronously in the background.

5.2 Event matching

Matching from an event to a large number of subscribers could be very inefficient if we use linear matching algorithm. Fortunately, our design can alleviate the problem to some extent. First, the matching load on a leaf node is bound by T, the subscription number threshold. Second, the property of subscription locality in our system has already filtered many irrelevant subscriptions for event matching, thereby minimizing the matching load system wide. Moreover, the subscription

locality property allows current subscription covering, merging and collapsing techniques, as a means of reducing event matching load, to be more effective. For example, Aguilera [12] proposed sublinear matching algorithms based on building a subscription tree that collapse similar subscriptions in order to reduce the matching load. Since similar subscriptions are clustered together on the leaf node, we argue that these techniques which exploit subscription relationships to reduce subscription management and event matching load would be more effective than the alternative design that distributes subscriptions randomly across nodes.

6 Event delivery

In this section, we start with embedded trees inherent in a DHT, then present event delivery algorithm that cleverly exploits the embedded trees in the underlying DHT to disseminate events, and finally give a brief discussion on the event delivery.

6.1 Embedded trees in a DHT

DHTs such as Chord [8], Pastry and Tapestry have inherent embedded trees formed by DHT links (or neighbor links). Here, we take Chord as a DHT example. In Chord, each node has a 160-bit ID, and the s nodes whose identifiers immediately follow a key are considered responsible for that key: they are the key's *successors*. The ID space in Chord wraps around such that zero immediately follow $2^{16} - 1$. Each Chord node (say, i) maintains a routing table: namely *finger table* and *successor list*. The finger table consists of the IP addresses and IDs of nodes which follow i at power-of-two distances in the identifier space (i.e., $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \cdots$). The successor list refers to i's immediate successors. In Chord, nodes consult their routing tables including *successor lists* and *finger tables* to route a message with a key k to a destination node whose ID is the successor of k. Consider each subscriber with a subscriber ID *sid*. The routing paths from a Chord node r (e.g., the DHT host node of a KDT leaf node A: $r = DHT\_host(A)$) to all these *sid*s (or subscribers) form a tree rooted at the node r, say $EmdTree_r$ (embedded tree rooted at r). As discussed below, the embedded tree will be used to disseminate events.

6.2 Event delivery using embedded trees

After event matching on a KDT leaf node (say, A) which has identified a list of matched subscribers (for simplicity, we use *sid*s to represent subscribers here), A will exploit the embedded tree $EmdTree_r$ rooted at A's host node $r = DHT\_host(A)$ to deliver events. Before starting event delivery, A first groups the matched subscribers according to r's neighbors including successors and finger nodes. Put another way, each subscriber s corresponds to a r's neighbor whose ID is *equal to or most immediately precedes* s's *sid*. This is based on the observation that when routing a message from r to s's node, r will forward the message to its neighbor whose ID is equal to or most immediately precedes s's node ID. As shown in right part of Fig. 3, subscribers x and y are grouped at r's neighbor f_2; subscribers z, w and v are grouped at r's neighbor f_3.

Event delivery is performed along the embedded tree $EmdTree_r$. Algorithm 1 and 2 outline the event deliver algorithm ($match\_set[1..k]$ corresponds to the current node's k neighbors and $match\_set[i]$ stores the subscriber IDs grouped at the i-th neighbor). The event delivery starts from r which sends out an event delivery message *carrying a corresponding subscriber ID list* along its neighbor links (as shown in Fig. 3). Upon receiving the message, each neighbor node (e.g., node s_2, f_2, or f_3) executes *route_message*(): If there is a subscriber ID matching its own ID, then it delivers the event to its local application/users; it also partitions the remaining subscriber IDs (if any) according to its own neighbor nodes (i.e., for each subscriber ID,

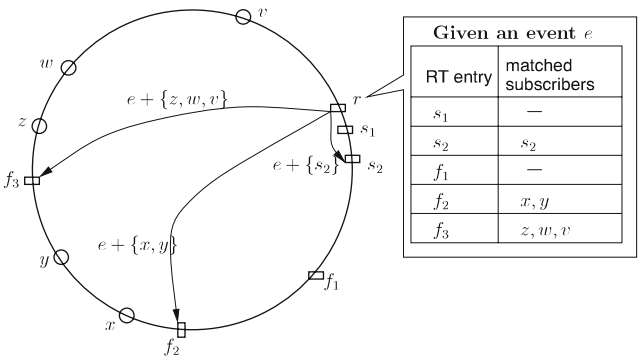


Fig. 3 Event delivery. r is the DHT host node for a KDT leaf node on the Chord ring. s_1 and s_2 are r's successors. f_1, f_2 and f_3 are r's finger nodes. s_2, x, y, z, w and v are subscribers for the event e. For ease of presentation, we use s_2, x, y, z, w and v to represent their subscriber IDs as well as their subscriptions here

choose a neighbor node whose ID is equal to or most immediately precedes the subscriber ID), and performs *deliver_event*() to deliver the messages each of which may carry a corresponding list of subscriber IDs to the remaining subscribers.

```
Algorithm 1: deliver_event(Event e,
               vector<ID> matched_set[1..k])
1) for i=1 to k
2)    if matched_set[i] is not empty
3)        Message M = e + match_set[i]
               //+ is a concatenation operator
4)        send M to the i-th neighbor node,
              which then calls route_message(M)
                upon receiving M
5)    endif
6) endfor


Algorithm 2: route_message(Message M)
1) vector<ID> matched_set[1..k]
2) Event e =  extract the event from M
3) vector<ID> list = extract the list of
              subscriber IDs from M
4) for each subscriber ID sid in list
5)    if sid == this node's ID
6)       deliver e to its local applications
         or users
7)    else
8)       find the i-th neighbor node whose node
              ID is equal to or most
              immediately precedes sid among
              all the neighbor nodes
9)       matched_set[i].push_back(sid)
10)   endif
11) endor
12) if matched_set is not empty
13)    deliver_event(e,matched_set)
14) endif
```

6.3 Discussion

The basic idea behind the event delivery algorithm is that all event delivery messages to those subscribers who share common ancestor nodes on the $EmdTree_r$ are aggregated into one single message along the path from the root r to their lowest common ancestor node, thus minimizing the number of messages. This event delivery is essentially a recursive process where each node along the dissemination paths of $EmdTree_r$ performs *deliver_event*() until the event reaches all subscribers.

The event delivery mechanism has several important features. First, it is a *match-first* approach: An event is first matched against the subscriptions in the KDT leaf node, generating subscriber ID lists each of which corresponds to a neighbor node of the leaf node's DHT host node. No subscription matching operation is performed along the dissemination paths except the embedded tree root node, due to the subscriber ID list contained in the message. Second, it exploits the embedded trees formed by the underlying DHT links to deliver events, eliminating nontrivial per-multicast tree construction and maintenance cost. To the best of our knowledge, we are the first to extensively yet smartly exploit the DHT overlay links to disseminate events. Exploiting DHT links allows some optimizations. E.g., the DHT link (or routing table) maintenance messages sent periodically can be piggybacked onto the event delivery messages to reduce the maintenance cost which is inherent and nontrivial in terms of bandwidth in a DHT. Finally, proximity neighbor selection (PNS) in the underlying DHT naturally ensures that event delivery on the embedded tree is proximity-aware.

However, encapsulating a subscriber ID list in an event delivery message may raise an issue if the subscriber ID list is undesirably long. We argue that our system can avoid this issue. First, the subscription number threshold T on the KDT leaf node gives the upper-bound of the list length. Second, the matched subscriber IDs on the KDT leaf node will be partitioned by its DHT host node's $O(\log N)$ neighbors if the subscribers are uniformly from the DHT overlay. Thus, the maximum size of the subscriber ID list contained in each event delivery message is $T/O(\log N)$. In addition, the subscriber ID list carried in each event delivery message is expected to be reduced by a factor of $O(\log N)$ at each step along the dissemination path. Finally, we propose a new technique, *one-hop subscription push* that could further make this a lesser issue. The basic idea is that a KDT leaf node A asks its DHT host node r to push the subscriptions corresponding to r's finger node f to f (as discussed earlier, a subscription is assigned to a node's neighbor whose ID is equal to or most immediately precedes the subscription's *sid*). The leaf node A then uses a *summary filter*[6] to represent the subscriptions pushed away. Upon an event e, A matches the event with the summary filter. If it is amatch, the node asks its DHT host node r

[6] A summary filter covers the subscriptions pushed away by exploiting covering relationships between subscriptions [31].

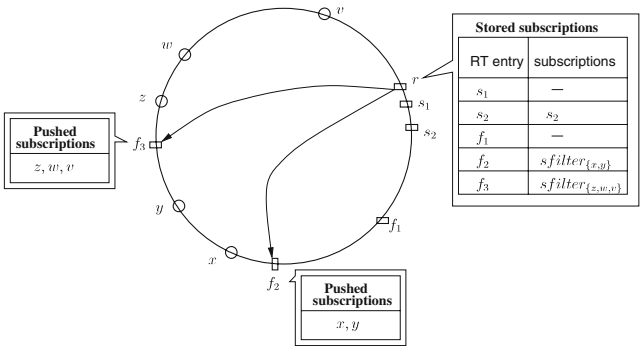


Fig. 4 One-hop subscription push. r is the DHT host node for a KDT leaf node on the Chord ring. s_1 and s_2 are r's successors. f_1, f_2 and f_3 are r's finger nodes. For simplicity, we use s_2, x, y, z, w and v to represent their corresponding subscriptions. *sfilter* represents the summary filter

to deliver e to the corresponding finger node (at this point, no subscription ID list is carried in the event delivery message) which in turn serves as an leaf node *agent* for those subscriptions pushed by r (or A) and starts delivering e to the matched subscribers. Figure 4 illustrates the one-hop subscription push. The leaf node asks its DHT host node r to push subscriptions $\{x, y\}$ and $\{z, w, v\}$ to r's finger nodes f_2 and f_3, respectively.

One-hop subscription serves two main purposes: (1) It reduces the maximum subscription ID list carried in an event delivery message and thus bandwidth cost. Note that with one-hop subscription push, no subscription ID list is carried from node r to its finger nodes f_2 and f_3. Only the event delivery messages sent from f_2 and f_3 will contain a subscriber ID list. As a result, the maximum size of the subscription ID list contained in the event delivery messages from f_2 and f_3 would be reduced to $T/O(\log^2 N)$ (The subscriber IDs is first split among r's $O(\log N)$ neighbors and then each such neighbor's $O(\log N)$ neighbors). (2) It allows a node to move part of its load (including subscription storage and event matching) to its neighbors for load balancing, if the neighbor node is lightly loaded or is willing to take some load. Note that all the load information can be piggybacked onto the DHT routing table maintenance messages sent periodically.

7 Evaluation

Our evaluation is based on a pub/sub scheme S for stock quotes application initially proposed in Meghdoot [7]. We focus our measurement effort on (1) KDT mechanism and the way it behaves under subscription installation and event publication and (2) performance and cost of event publication and event delivery.

7.1 Experimental setup

We implemented our system based on **p2psim 3.0**. P2psim can simulate Chord and it does not simulate link transmission rate or queuing delay. The number of successor nodes for each Chord node is 16 and the finger table base is 2. We set both finger stabilization interval and successor stabilization interval to be 72 seconds. We also configured Chord with proximity neighbor selection (PNS) which allows each Chord node to choose physically close nodes as routing table entries to improve lookup latency.

The simulated network consists of 1024 nodes with inter-node latencies derived from measuring the pairwise latencies of 1024 DNS servers on the Internet using King method. The average round-trip time is 152ms. Unless otherwise specified, our experimental results presented in this paper are based on this simulated network.

Simulations were initialized with one node. A new node joins the system at a randomly chosen time, until the total number of nodes reaches the bound (e.g., 1024 nodes). After system stabilization, we scheduled subscription installation events which create a KDT

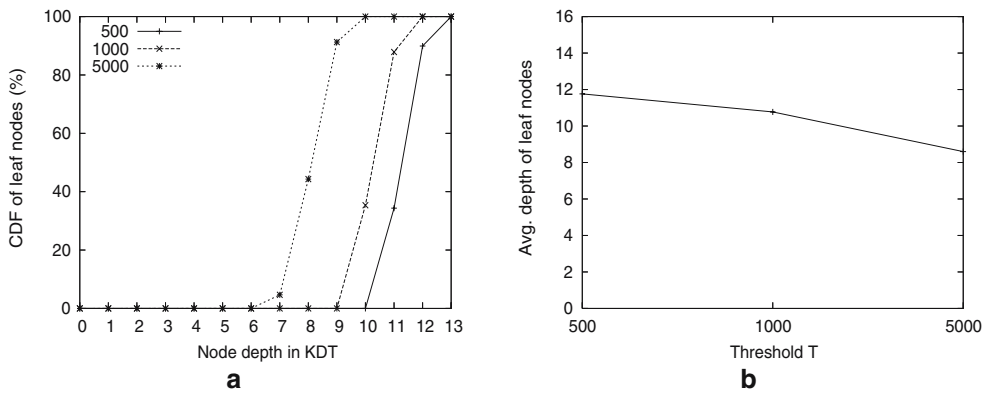


Fig. 5 **a** CDF of leaf node depth for a KDT with 1 million subscriptions and T of 500, 1000 and 5000 respectively. **b** Variation in tree depth as a function of T with 1 million subscriptions

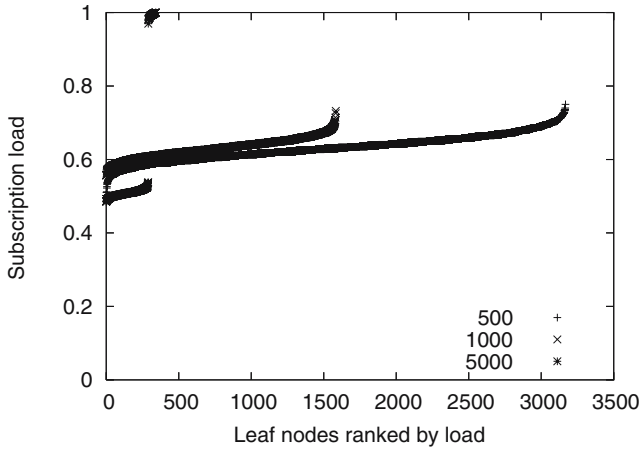


Fig. 6 Subscription load in the KDT leaf nodes for varying Ts. The number of subscriptions is 1 million

and dynamically expand it on top of Chord as subscriptions are installed. After subscription installation, we simulated event publication which triggers event delivery across the system.

The pub/sub scheme S for the stock quotes application is defined as $S = \{(Date, string, 2/Jan/98, 31/Dec/02), (Symbol, string, \text{“}aaa\text{”}, \text{“}zzzzz\text{”}), (Open, float, 0, 500), (Close, float, 0, 500), (High, float, 0, 500), (Low, float, 0, 500), (Volume, integer, 0, 310000000)\}$. Specifically, *Symbol* is stock name. *Open* and *Close* are opening and closing prices for a stock on a given day. *High* and *Low* are the highest and lowest prices for the stock on that day. *Volume* is the total amount of trade in the stock on that day. Given the scheme S, an example subscription s is $(123456, (Symbol = \text{“}yhoo\text{”}) \wedge (High > 35.23))$, subscribed by a subscriber with $sid = 123456$ for events on the stock of *Yahoo* when its highest price is greater than \$35.23.

We generated subscriptions by using five template subscriptions suggested in Meghdoot with different probabilities. The five templates are $T_1 = \{(Symbol = P_1) \wedge (P_2 \leq Open \leq P_3)\}$ with probability 20%, $T_2 = \{(Symbol = P_1) \wedge (Low \leq P_2)\}$ with probability 35%, $T_3 = \{Symbol = P_1 \wedge (High \geq P_2)\}$ with probability 35%, $T_4 = \{Symbol = P_1 \wedge (Volume \geq P_2)\}$ with probability 5%, and $T_5 = \{Volume \geq P_1\}$ with probability 5%. The templates with general interests (e.g.,T_4 and T_5) are assigned low probabilities due to the fact that in a real application subscribers are usually interested in specific events related to their narrow interests. The number of stocks and subscriptions used in simulations were 100 and 10^6 respectively by default, unless otherwise specified. The events were generated randomly from S and we used 10^5 events in simulations.

7.2 Understanding KDT

In the first set of experiments, we measured the properties of a KDT using 1 million subscriptions and 10^5 events. We used three metrics: (1) Tree depth. (2) Subscription load (number of subscriptions per KDT leaf node as a fraction of the subscription number threshold T). (3) Access load (number of DHT accesses per tree level as a percentage of the total DHT accesses)

Tree Depth: Figure 5a depicts the CDF of the depth of leaf nodes in a KDT for varying Ts. Figure 5b shows the variation in average depth of the KDT with respect to T. We can see that the tree depth decreases with T, i.e., larger Ts result in shallower trees.

Subscription Load: The purpose of this experiment is to explore (1) how full the leaf nodes are as a fraction of T and (2) the variation in subscription load among the leaf nodes. Figure 6 shows the subscription load among the leaf nodes for varying Ts. Two main observations can be made from this figure. First, smaller Ts exhibit more load balanced among the leaf nodes. Second, larger Ts (i.e., 5000) result in higher utilization as a fraction of T in some leaf nodes.

Access Load: Unlike traditional linked-based tree structure, KDT lookups can bypass the root and start from the lower levels in the tree. This experiment is to look at the access load on the nodes by tree level

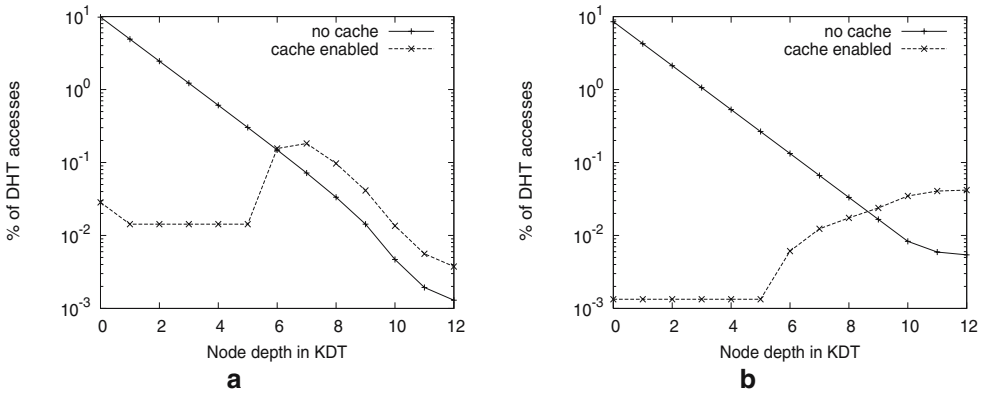


Fig. 7 **a** DHT accesses to a KDT at each tree level for installing 1 million subscriptions where $T = 1000$. **b** DHT accesses to a KDT at each tree level for publishing 10^5 events where $T = 1000$

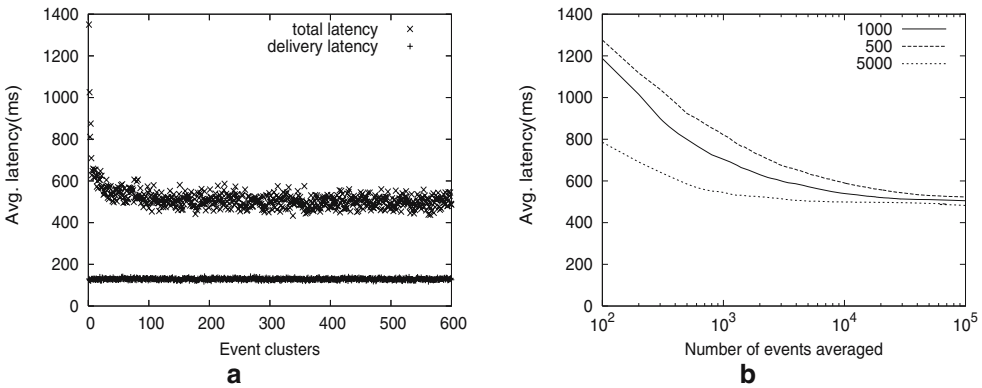


Fig. 8 **a** Plot of total latency and event delivery latency for each event cluster when publishing 10^5 events. $T = 1000$. **b** Plot of total latency for varying T when publishing 10^5 events

with and without the client-side KDT-lookaside cache enabled. As discussed earlier, using lookaside cache at client-side allows us to avoid having the upper levels of the tree be potential bottleneck. Figure 7a and b show the spread of DHT accesses across KDT levels for subscription installation and event publication respectively. When the cache is enabled, the client (subscriber or publisher) starts with an empty cache and continuously fills the cache if necessary after an operation of subscription installation or event publication. From the graphs, we can see that the look-aside cache could effectively avoid the hotspots at the levels close to the root. Note that the bulk of activities for subscription installation occur in the tree depth of 6 to 9 while the bulk of activities for event publication occur in lower levels of tree and leaf nodes. This is because events were generated uniformly from the event space and a few number of events can quickly fill the cache with the current shape of the KDT, avoiding accessing the higher levels of the KDT for successive event publications. However, in subscription installation, the dynamic expansion of the KDT and possible splits of subscriptions during installation prevent the installation operations from starting from the lower levels of the KDT.

7.3 Event publication and delivery

In the second set of experiments, we investigated the event publication and delivery latency. The client-side cache is enabled. We pre-loaded the KDT with 1 million subscriptions. After installing the subscriptions, we started the client (or publisher) with an empty lookaside cache. Then, the publisher continuously published 10^5 events into the system, each of which was delivered to the matched subscribers. The percentage of nodes as subscribers per event is about 25%. We used *total latency* to represent the latency of event publication and delivery. Event-matching processing delay on the leaf node is not counted. Figure 8a depicts total latency and event delivery latency for the first 30, 000 events. Each data point is the average number of every 50 events in the order of arrival times. For example, the first event cluster represents the earliest arrived 50 events. Two main observations can be made: (1) The client-side lookaside cache effectively improves the total latency. E.g., after about 1, 500 events, the total latency for the remaining events is almost flat. This is because the lookaside cache effectively reduces event publication latency. (2) Event delivery is very efficient, incurring a latency of 119ms.

Figure 8b shows the total latency for varying Ts. The x-axis represents the number of events considered (by their arrival times) for computing the average total latency. Again, the lookaside cache is very effective in reducing total latency, that is, by improving event publication latency. In addition, larger Ts result in shallower trees and thus reduce total latency significantly, especially when the lookaside cache has not yet been filled with the shape of the KDT. When the cache is filled with the shape of the KDT, there is not much difference in total latency among different Ts.

Table 1 Results for varying percentages of nodes as subscribers per event

Metric	5%	10%	20%	30%	40%	50%	60%	70%	80%
Latency(ms)	119	119	119	119	119	119	119	119	119
BW_cost(Bytes/node)	60.75	62.73	64.82	65.96	66.67	67.17	67.54	67.82	68.03
Overhead	0.86	0.53	0.28	0.18	0.12	0.08	0.05	0.04	0.02

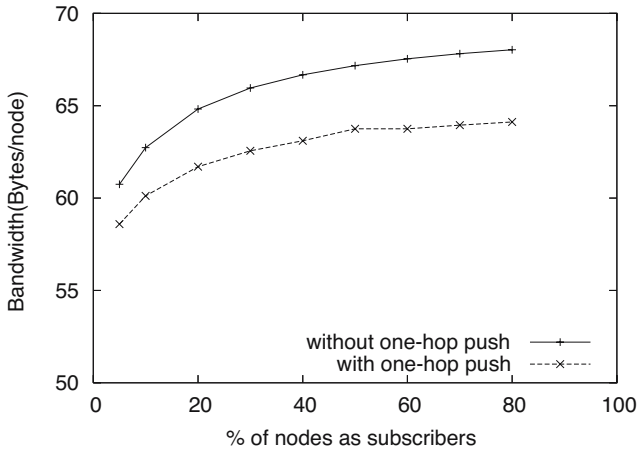


Fig. 9 Bandwidth cost per event delivery w/o one-hop subscription push

7.3.1 Event delivery

In this set of experiments, we measured the performance and cost of event delivery, by using the following metrics: (1) Latency. (2) Overhead: It is defined as the ratio of the number of intermediate nodes (non-subscriber nodes) involved during the delivery of an event to the number of subscribers for this event. The lower the overhead, the more efficient is the event delivery. Low overhead results from message aggregation in event delivery. (3) Bandwidth cost: It is defined as ratio of the total bandwidth cost incurred by an event delivery to the number of nodes involved (including the intermediate nodes and subscriber nodes). The size in bytes of each event delivery message is counted as 20 bytes for headers, 33 bytes for the event, and 4 bytes for each subscriber ID carried in the message.

Table 1 summarizes the results of event delivery for varying numbers of subscribers as a percentage of 1024 nodes. One-hop subscription push is not used. For a given number of subscribers, the number of events is 10^5. As the number of subscribers per event increases, the delivery latency keeps constant at 119ms while the bandwidth cost increases modestly due to the increased size of the subscriber ID list carried in the event delivery message. However, the overhead drops dramatically as the number of subscribers per event increases. This shows that the event delivery mechanism can efficiently deliver events to a large number of subscribers, involving only a small number of intermediate (or non-subscriber) nodes by its message aggregation along dissemination paths.

Figure 9 shows the bandwidth cost w/o one-hop subscription push for event delivery. As mentioned earlier, one-hop push eliminates the subscriber ID list contained in the event delivery messages sent from the leaf nodes and could decrease the initial subscriber ID list by a factor of $O(\log^2 N)$, thereby reducing the bandwidth cost. Note that the reduction in bandwidth cost is per event. A small reduction could lead to huge reduction in overall bandwidth consumption across the system.

Performance Comparison. We also compared PRESS against Meghdoot [7] in terms of event delivery performance because event delivery is critical to a pub/sub system. We used two metrics: (1) CDF of event distribution with respect to the percentage of nodes visited per event (which measures the cost of event delivery), and (2) event delivery load, defined as the ratio of event messages a peer receives to the total number of messages processed in the system (which measures event delivery load distribution). However, we admit the comparison is by no means complete. In our next step, we plan to develop a more detailed Meghdoot simulator and compare the two systems more thoroughly.

Figure 10a shows CDF of event distribution with respect to the number of nodes visited during event delivery. The *x*-axis represents the percentage of nodes visited to deliver an event out of the total number of nodes in the system. PRESS shows better performance

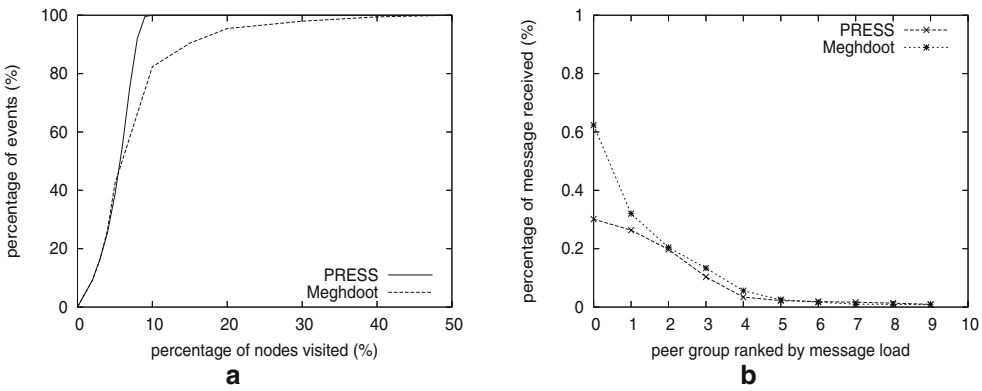


Fig. 10 Comparisons in a 1024-node system with 10^4 subscriptions and 10^5 events. **a** CDF of event distribution with respect to the number of nodes visited. **b** Distribution of event delivery load by peer group

than Meghdoot in that all event deliveries end up with visiting at most 10% nodes. The is mainly due to message aggregation during event delivery. Event delivery load measures message load imposed on a node during event delivery. We sorted the peer nodes in decreasing order of the load and grouped them by their rank into group size 10% each. Figure 10b shows the average load on each group. The load distribution among peers is more balanced in PRESS than Meghdoot. For example, the maximum load on a node in PRESS is about 0.3% of the total messages, which is very good. This shows that PRESS is able to fairly distribute event delivery load among the nodes in the system.

8 Conclusions

PRESS uses KD-trees to dynamically partition and organize subscriptions, thereby preserving subscription locality, minimizing event matching load, and ensuring load balance among nodes. PRESS exploits the embedded trees in the underlying DHT for event dissemination, thereby imposing little overhead on the DHT. Via simulations, we have showed that PRESS can deliver events to a large number of subscribers at low overhead and latency while incurring modest bandwidth cost. The framework can be used to support multiple content-based pub/sub systems. For example, we can layer the KDT corresponding to a pub/sub scheme S on top of a DHT by planting each KDT node (with label L) in a DHT node which is responsible for the key $k = h(S, L)$. Due to the uniformity of the hashing function, the KDTs (corresponding to the multiple pub/sub schemes) span different DHT nodes with high probability, thereby ensuring load balance across DHT nodes.

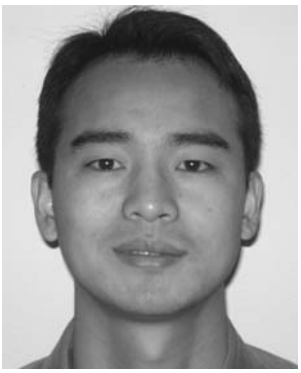
PRESS currently does not well support underspecified (or wild-card) attributes in the subscription space. For example, in an extreme case, all the subscriptions are of the form $A_1 > c_1$, though this might be barely true in real applications. It would be difficult to split the subscription space into disjoint sub-spaces. We are currently investigating techniques to address this issue. One possible solution we are considering is to use a "hybrid" KDT tree: upon this extreme case, the KDT node may split the subscriptions randomly into its two child nodes; during event matching, an event entering the KDT node will be multicast to its two child nodes. One alternative is to let the KDT internal nodes maintain such wild-card subscriptions to avoid subscription partitioning, and then events are published to not only the relevant leaf nodes but also the relevant internal nodes.

Acknowledgements We thank Adair Dingle for her valuable comments and inspiring discussions on the first draft of this paper. We also thank anonymous reviewers for their constructive feedbacks which helped improve the paper.

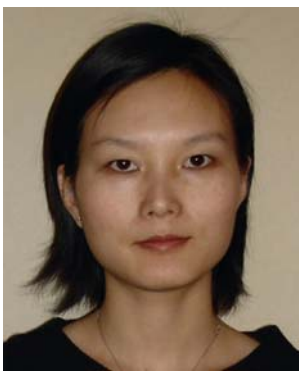
References

1. Fabret F, Jacobsen HA, Llirbat F, Pereira J, Ross KA, Shasha D (2001) Filtering algorithms and implementation for very fast publish/subscribe systems. In: Proceedings of the 2001 ACM SIGMOD, vol. 30. Santa Barbara, CA, pp 115–126
2. Segall B, Arnold D (1997) Elvin has left the building: a publish/subscribe notification service with quenching. In: Proceedings of AUUG. Brisbane, Australia, pp 243–255, September
3. Triantafillou P, Economides A (2004) Subscription summarization: a new paradigm for efficient publish/subscribe systems. In: Proceedings of the 24th IEEE ICDCS. Tokyo, Japan, pp 562–571, March
4. Carzaniga A, Wolf AL (2003) Forwarding in a content-based network. In: Proceedings of ACM SIGCOMM. Karlsruhe, Germany, pp 163–174, August
5. Carzaniga A, Rutherford MJ, Wolf AL (2004) A routing scheme for content-based networking. In: Proceedings of IEEE INFOCOM. Hongkong, China, pp 918–928, March
6. Cao F, Singh JP (2005) MEDYM: match-early and dynamic multicast for content-based publish-subscribe service networks. In: Proceedings of the 4th international workshop on distributed event-based systems. Washington, DC, pp 370–376
7. Gupta A, Sahin OD, Agrawal D, Abbadi AE (2004) Meghdoot: content-based publish/subscribe over P2P networks. In: ACM/IFIP/USENIX 5th international middleware conference. Toronto, Ontario, Canada, October
8. Stoica I, Morris R, Karger D, Kaashoek M, Balakrishnan H (2001) Chord: a scalable peer-to-peer lookup service for internet applications. In: Proceedings of ACM SIGCOMM. San Diego, CA, pp 149–160, August
9. Rowstron A, Druschel P (2001) Pastry: scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Proceedings of the 18th IFIP/ACM international conference on distributed system platforms (Middleware). Heidelberg, Germany, pp 329–350, November
10. Zhao BY, Kubiatowicz JD, Joseph AD (2001) Tapestry: an infrastructure for fault-tolerance wide-area location and routing. Tech. Rep. UCB/CSD-01-1141, Computer Science Division, University of California, Berkeley, April
11. Ratnasamy S, Francis P, Handley M, Karp R, Shenker S (2001) A scalable content-addressable network. In: Proceedings of ACM SIGCOMM. San Diego, CA, pp 161–172, August
12. Aguilera MK, Strom RE, Sturman DC, Astley M, Chandra TD (1999) Matching events in a content-based subscription system. In: Proceedings of the 8th ACM symposium on principles of distributed computing (PODC). Atlanta, GA, pp 53–61, May
13. Li G, Hou S, Jacobsen H-A (2005) A unified approach to routing, covering and merging in publish/subscribe systems based on modified binary decision diagrams. In: Proceedings of international conference on distributed computing systems (ICDCS), Columbus, OH, June
14. Banavar G, Chandra T, Mukherjee B, Nagarajarao J, Strom RE, Sturman DC (1999) An efficient multicast protocol for

content-based publish-subscribe systems. In: Proceedings of the 19th IEEE ICDCS. Washington, DC, pp 262–272, June
15. Carzaniga A, Rosenblum DS, Wolf AL (2001) Design and evaluation of a wide-area event notification service. ACM Trans Comput Syst 19(3):332–383
16. Banavar G, Chandra T, Mukherjee B, Nagarajarao J, Strom RE, Sturman DC (1999) An efficient multicast protocol for content-based publish-subscribe systems. In: Proceedings of the 19th IEEE international conference on distributed computing systems(ICDCS). Austin, TX, pp 262–272, May
17. Cao F, Singh JP (2004) Efficient event routing in content-basedpublish/subscribe service networks. In: Proceedings of INFOCOM, vol. 2. Hong Kong, China, pp 929–940, March
18. Zhang R, Hu YC (2005) HYPER: a hybrid approach to efficient content-based publish/subscribe. In: Proceedings of international conference on distributed computing systems (ICDCS). Columbus, OH, June
19. Rowstron AIT, Kermarrec A-M, Castro M, Druschel P (2001) SCRIBE: the design of a large-scale event notification infrastructure. In: Proceedings of the 3rd international networked group communication, pp 30–43
20. Zhuang SQ, Zhao BY, Joseph AD, Katz RH, Kubiatowicz J (2001) Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination. In: Proceedings of the eleventh international workshop on network and operating system support for digital audio and video (NOSSDAV). Port Jefferson, New York, pp 11–20, June
21. Castro M, Druschel P, Kermarrec A-M, Nandi A, Rowstron A, Singh A (2003) Splitstream: high-bandwidth multicast in cooperative environments. In: Proceedings of the 19th ACM symposium on operating systems principles (SOSP). Bolton Landing, NY, October
22. Tam D, Azimi R, Jacobsen H-A (2003) Building content-based publish/subscribe systems with distributed hash tables. In: Proceedings of the international workshop on databases, information systems and peer-to-peer computing. Berlin, Germany, September
23. Terpstra WW, Behnel S, Fiege L, Zeidler A, Buchmann AP (2003) A peer-to-peer approach to content-based publish/subscribe. In: Proceedings of the second international workshop on distributed event-based systems (DEBS). San Diego, CA, June
24. Triantafillou P, Aekaterinidis I (2004) Content-based publish-subscribe over structured P2P networks. In: Proceedings of the third international workshop on distributed event-based systems (DEBS). Edinburgh, Scotland, UK, pp 104–109, May
25. Perng G, Wang C, Reiter MK (2004) Providing content-based services in a peer-to-peer environment. In: Proceedings of the third international workshop on distributed event-based systems (DEBS). Edinburgh, Scotland, UK, pp 74–79, May
26. Choi Y, Park K, Park D (2004) HOMED: a peer-to-peer overlay architecture for large-scale content-based publish/subscribe systems. In: Proceedings of the third international workshop on distributed event-based systems (DEBS). Edinburgh, Scotland, UK, pp 20–25, May
27. Zhu Y, Hu Y (2007) Ferry: an P2P-based architecture for content-based publish/subscribe services. IEEE Trans Parallel Distrib Syst 18(5):672–685
28. Bentley JL (1975) Multidimensional binary search trees used for associative searching. Commun ACM 18(9):509–517
29. Zhang C, Krishnamurthy A, Wang RY (2005) Brushwood: distributed trees in peer-to-peer systems. In: Proceedings of 4th international workshop on peer-to-peer systems (IPTPS), Ithaca, NY, February
30. Chawathe Y, Ramabhadran S, Ratnasamy S, LaMarca A, Shenker S, Hellerstein J (2005) A case study in building layered DHT applications. In: Proceedings of SIGCOMM. Philadelphia, PA, pp 97–108, August
31. Wang Y-M, Qiu L, Achlioptas D, Das G, Larson P, Wang HJ (2002) Subscription partitioning and routing in content-based publish/subscribe systems. In: Proceedings of the 16th international symposium on distributed computing (DISC). Toulouse, France, October



Yingwu Zhu received the PhD degree in Computer Science & Engineering from University of Cincinnati in 2005. He received the BS and MS degrees in Computer Science from Huazhong University of Science & Technology, at Wuhan, China, in 1994 and 1997, respectively. He is an assistant professor of Computer Science and Software Engineering at Seattle University. His research interests include operating systems, storage systems, peer-to-peer computing, distributed systems, and computer networks.



Haiying Shen received the BS degree in Computer Science and Engineering from Tongji University, China in 2000, and the MS and Ph.D. degrees in Computer Engineering from Wayne State University in 2004 and 2006, respectively. She is currently an Assistant Professor in the Department of Computer Science and Computer Engineering of University of Arkansas. Her research interests include distributed and parallel computer systems and computer networks, with an emphasis on peer-to-peer and content delivery networks, mobile computing, high performance cluster and grid computing. She is a member of IEEE and ACM.