Ferry: A P2P-Based Architecture for Content-Based Publish/Subscribe Services [*]

Yingwu Zhu
Department of CSSE, Seattle University
zhuy@seattleu.edu

Yiming Hu
Department of ECECS, University of Cincinnati
yhu@ececs.uc.edu

Abstract

We propose Ferry, an architecture that extensively yet wisely exploits the underlying distributed hash table (DHT) overlay structures to build an efficient and scalable platform for content-based publish/subscribe (pub/sub) services. Ferry aims to host any and many content-based pub/sub services: any pub/sub service with a unique scheme can run on top of Ferry, and multiple pub/sub services can coexist on top of Ferry. For each pub/sub service, Ferry does not need to maintain or dynamically generate any dissemination tree. Instead, it exploits the embedded trees in the underlying DHT to deliver events, thereby imposing little overhead. Ferry can support a pub/sub scheme with a large number of event attributes. To deal with skewed distribution of subscriptions and events, Ferry uses one-hop subscription push and attribute partitioning to balance load.

Indexed Terms: DHT, subscription installation, subscription management, event delivery, one-hop subscription push, content-based publish/subscribe.

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 expressive subscriptions which specify complex filtering criteria by using a set of predicates over event attributes. Upon receiving an event

[*] Extended version of the work presented in Proceedings of ICPP'05. **Nominated for Best Paper Award**.

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.

1.1 Content-based Pub/Sub Model

Fabret et al. [1] proposed a content-based pub/sub scheme 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 $type$ 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 attributes $\in S$ and it can be represented as $e = \{A_1 = c_1, A_2 = c_2, \cdots, A_n = c_n\}$. A predicate has a *name*, *type*, *operator* and *value* and is used to specify a constant value or range for an attribute. A subscription is a conjunction of predicates over one or more attributes. If a subscription needs to specify multiple predicates over the same attribute, it can be modeled as a combination of multiple subscriptions, each of which specifies one value or continuous range over the attribute. For simplicity of presentation, we assume each subscription specifies a value or continuous range over 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.

1.2 Motivation

Current content-based pub/sub systems are either centralized or distributed. One example of centralized systems is Elvin [2]. Elvin uses a central server that stores all the subscriptions, evaluates the subscriptions upon events and delivers events to the matched subscribers. Centralized solutions, while simple, 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. However, restrictions have to be placed on subscriptions such that they must contain at least one equality predicate, sacrificing flexibility and expressiveness of subscriptions.

Many distributed pub/sub systems [3, 4, 5, 6, 7, 8, 9] have been proposed by using routing trees to perform event delivery based on multicast techniques. Siena [4] builds a symmetric spanning tree and each pub/sub server can be a publisher or subscriber. Gryphon [9] organizes the pub/sub network into a single-source tree and proposes a link matching algorithm to forward events towards directions of matching subscriptions.

As distributed hash tables (DHTs) [10, 11, 12, 13] attract more and more interests from both research and industrial communities due to their scalability, fault-tolerance and self-organization, we have seen many attempts in building and designing DHT-based pub/sub systems [14, 15, 16, 17, 18, 19, 20, 21, 22]. In such systems, peers cooperate in storing subscriptions and routing events to subscribers in a fully distributed manner.

However, all these distributed and DHT-based solutions suffer some or all of the following limitations: (1) Using explicitly constructed multicast trees for event delivery, introduces non-trivial cost (e.g., bandwidth consumption) in tree construction and maintenance, especially in dynamic peer-to-peer systems where nodes join or leave at will. (2) System such as Scribe [21] and Bayeux [22] are essentially topic-based pub/sub systems. They do not directly support content-based pub/sub services. (3) Some solutions (e.g., [18]) place some restrictions on subscriptions, thus sacrificing expressiveness of subscriptions which however is one major feature distinguishing content-based pub/sub from topic-based pub/sub [23, 24]. (4) With few exceptions [14], most of them fail to address the load-balancing issue while subscription and event distributions in real-world applications are highly non-uniform.

1.3 Goals

Challenges for content-based pub/sub systems include efficient subscription management and event matching, load balancing, and efficient and scalable event delivery. In this paper, we propose Ferry, a *fully distributed, efficient and scalable architecture for content-based pub/sub services* built on top of DHTs. Ferry serves as a platform to host any and many content-based pub/sub services. In particular, the goals of Ferry which make our contributions, are as follows:

1. *Design novel subscription installation and management algorithms that facilitate aggregation of event delivery messages, thereby minimizing the number of messages across the system.*

2. *Propose one-hop subscription push and attribute partitioning to address the load balancing issue in real-world pub/sub applications.*

3. *Propose an efficient and scalable event delivery algorithm that is virtually maintenance-free by making wise use of embedded trees in the underlying DHT.*

To the best of our knowledge, Ferry is the *first* solution that *extensively yet smartly* exploits the DHT overlay links to manage subscriptions and disseminate events for content-based pub/sub services. Its load-balancing technique, one-hop subscription push, also makes wise use of the DHT links. By exploiting DHT links in its design, Ferry have numerous advantages: (1) The fault-tolerance and self-organizing nature of DHT links makes Ferry resilient to node failures. (2) Ferry does not construct or maintain multicast trees for event delivery, and it is virtually maintenance-free. (3) Grouping subscriptions along DHT links in subscription management facilitates message aggregation during event delivery, thus minimizing the number of messages across the system. (4) The proximity neighbor selection (PNS) property of DHT links naturally enables proximity-aware event delivery along the DHT links, yielding good delivery performance. (5) The DHT routing table maintenance messages (sent periodically by the underlying DHT) could be piggybacked onto the event delivery messages to reduce the DHT maintenance cost that is nontrivial in terms of bandwidth.

We have built Ferry on top of **p2psim**, a discrete-event packet level simulator. Via detailed simulations, we have evaluated Ferry extensively in terms of overlay hops, latency, overhead, and bandwidth cost. We show that Ferry can deliver events to various numbers of subscribers under different network sizes efficiently and timely. We have also compared performance of Ferry and Meghdoot [14] in event delivery, showing Ferry has a better performance.

The rest of the paper is structured as follows. Section 2 provides a survey of related work. Section 3 gives necessary background. We present the design of Ferry in Section 4. Section 5 discusses an alternative design and addresses its limitations. Section 6 presents experimental setup and results. We conclude the paper in Section 7.

2 Related Work

Pub/sub systems are mainly categorized into *topic-based* and *content-based*. In topic-based systems, each publisher and subscriber join the groups containing the topics they are interested in and events that belong to a topic are broadcasted to all subscribers of the corresponding group. The example systems include ISIS [23] and iBus [24].

Content-based pub/sub systems are preferable as they allow subscribers to specify their interests in a fine-grained way, i.e., a subscriber can express his/her interests by a set of predicates over event attributes. Many distributed content-based pub/sub systems [3, 4, 5, 6, 7, 8, 9] have been proposed by using routing trees to deliver events to the subscribers based on multicast techniques. Among them, Ferry is most similar to MEDYM [8]. In MEDYM, each node can be a *matcher* for some subscriptions and events. Upon receiving an event, some matcher responsible for this event matches the 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. However, Ferry differs from MEDYM in that it exploits the embedded trees inherent in the underlying DHT to deliver events, thereby imposing little overhead.

Many attempts have been made in designing a P2P-based pub/sub system [14, 15, 16, 17, 18, 19, 20, 21, 22]. Scribe [21] and Bayeux [22] are essentially a topic-based pub/sub system built on top of Pastry and Tapestry, respectively. They do not directly support content-based pub/sub services. SplitStream [25] is an application-level multicast system built from Scribe for high-bandwidth data dissemination, by splitting content into k stripes each of which corresponds to a Scribe multicast tree. Recent work [26] has shown that the maintenance cost of multicast trees due to non-DHT links (40% or more) in Scribe/SplitStream is non-trivial under node churn, while Ferry does not impose any such maintenance cost by exploiting the embedded trees in DHTs. Tam et al. [18] proposed a content-based pub/sub system built from Scribe. The problem with their system is that it has some restrictions on the expression of subscriptions and thus sacrifices expressiveness of subscriptions.

Terpstra et al. [17] proposed a content-based pub/sub system built on top of Chord. In this system, both filter updates and event routing actually use a broadcasting algorithm. Triantafillou et al. [16] also built their content-based pub/sub system on top of Chord. A subscription is stored in nodes with 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. However, the main drawback is that subscription installation and update are expensive due to the large number of nodes and messages potentially involved.

Reach [19] 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 [15] maintains a semantic overlay where each node's identifier is derived from its subscriptions. However, they have the following 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. Finally, using a bit vector to express user's interests in Reach is not fine-grained, while in HOMED, it may be difficult to derive node IDs from their subscriptions while preserving high expressiveness of subscriptions.

Meghdoot [14] is based on CAN [13]. Subscriptions are stored on a zone according to the coordinate determined by event attribute values. Considering skewed distributions of both 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 structure is determined by the pub/sub scheme and the overlay dimensionality is proportional to the number of event attributes.

3 Background: Chord

Chord [10], a representative DHT, basically supports one operation $lookup(key)$ which maps a 160-bit key to a set of IP addresses of the nodes responsible for the key. In Chord, each node has a 160-bit identifier, and the s nodes whose identifiers immediately follow a key are considered responsible for that key: they are the key's *successors*. To provide reliable lookup even if half of the nodes fail in a 2^{16}-node network, the number of successors, s, is 16 in the Chord implementation. The ID space in Chord wraps around such that zero immediately follows $2^{16} - 1$ [27].

The *base* Chord lookup algorithm works as follows. Each Chord node (say, i) maintains a routing table: namely a *finger table* and a *successor list*. The finger table consists of the IP addresses and IDs of nodes which follow the Chord node i at power-of-two distances in the identifier space (i.e.,$\frac{1}{2}$,$\frac{1}{4}$,$\frac{1}{8}$, $\frac{1}{16}$, ...). The successor list refers to i's s immediate successors. When a node issues a lookup with a key k, it consults a sequence of other nodes, asking each in turn which node to talk to next; each node in this sequence answers with the node from its finger table whose ID most immediately precedes k. By $O(\log N)$ consultations, the originating node will find the key k's *predecessor node,* and then it request the predecessor node for its successor list, which is the result of the lookup. Note that this style of lookup is called *iterative* since the originating node controls each step of the lookup. Dabek et al. [27] have shown that *recursive* lookups have lower latencies than iterative lookups. Hence, Ferry uses recursive lookups: each node in the lookup path directly forwards the query to the next node.

4 System Design

In this section, we first present system overview in Section 4.1 and then detail Ferry's design in Section 4.2-4.5. Section 4.6 discusses load balancing issues and Section 4.7 addresses Ferry's scalability to the number of event attributes. We provide a discussion of Ferry in Section 4.8 and demonstrate Ferry's adaptability to other DHTs in Section 4.9.

4.1 Overview

We base the design of Ferry on Chord. However, the techniques discussed here are applicable or easily adaptable to other DHTs such as Pastry [11] and Tapestry [12]. Ferry aims to serve as a platform to host multiple pub/sub services with unique schemes. For ease of exposition, we base our discussion on a pub/sub scheme $S = \{A_1, A_2, ..., A_n\}$.

Ferry is essentially a rendezvous network built on top of Chord to support content-based pub/sub services. In Ferry, each node could serve as a rendezvous point (RP) for some subscriptions and events, and as an intermediate node on the paths of event delivery. Given a pub/sub scheme $S = \{A_1, A_2, ..., A_n\}$, the RP nodes for its subscriptions and events are the most immediate successors of $k_i = h(A_i)$, where k_i is a key derived from an attribute A_i by using the consistent hash function $h()$ which is used in Chord to produce node IDs and data keys. Subscriptions and events entering the system are routed to their corresponding RP nodes. A subscription in Ferry is stored on a RP node in the form of a pair (sid, p), where sid is the subscriber's node ID (*subscriber ID* for short) and p is a conjunction of predicates specifying the subscriber's interests (e.g., $p = \{(A_1 = c_1) \wedge (c_2 \leq A_3 \leq c_3)\}$). When a node wants to publish an event, the event is first directed to the RP nodes where the event is matched to the subscriptions. Once those subscriptions matching the event are identified, the event is then delivered to the corresponding subscribers by using Ferry's novel event delivery algorithm.

Ferry is based on four key mechanisms: (1) Subscription Installation (Section 4.2), (2) Subscription Management (Section 4.3), (3) Event Publication and Matching (Section 4.4) and (4) Event Delivery (Section 4.5).

4.2 Subscription Installation

When a user wishes to subscribe for some events, the user first has to register his/her interests to a RP node in the form of subscription $s = (sid, p)$. Ferry explores two installation algorithms, called *RndRP* and *PredRP*, respectively. As shown in Algorithm 1, the primary purpose of RndRP is to *uniformly at random* distribute subscriptions over the RP nodes of the scheme S, even in the face of skewed subscription distribution, i.e., some subscriptions are very popular. However, RndRP could result in inefficient event delivery. Consequently, we propose a more efficient algorithm, PredRP, for subscription installation in Ferry.

Algorithm 1 $RndRP$(Subscription s)

```
$A_i \leftarrow$ randomly choose an attribute from $S$
$k = h(A_i)$ //$h$ is a consistent hash function used by Chord
store $s$ in a RP node which is an immediate successor node of $k$
```

Algorithm 2 outlines the PredRP algorithm. The basic idea behind PredRP is that *a subscription* s *is stored in a RP node whose node ID is equal to or most immediately precedes* s*'s subscriber ID among all the RP nodes of the scheme* S. Figure 1 illustrates RndRP and PredRP. In RndRP, the RP nodes r_1 and r_2 each may store the subscriptions from the subscribers distributed over the whole Chord ring space and thus the event delivery messages may need to traverse the whole Chord ring space. However, with PredRP, the event delivery messages from RP nodes r_1 and r_2 only need to traverse a fraction of the Chord ring space due to the fact that each RP node stores only those subscriptions from a non-overlapped, contiguous region of the Chord ring space (e.g., r_1 is responsible for the subscriptions from the range $[r_1, r_2)$, and r_2 is responsible for the subscriptions from the range $[r_2, r_1)$). As will be shown in Section 6, PredRP achieves better performance than RndRP by avoiding sending the redundant messages across the Chord ring space and making the messages to traverse shorter distance of the Chord ring.

Algorithm 2 $PredRP$(Subscription s)

```
choose an attribute $A_i$ from $S$ such that $h(A_i)$ either is equal to or most immediately precedes $s$'s subscriber ID
    among all attributes
$k = h(A_i)$
store $s$ in a RP node which is an immediate successor node of $k$
```

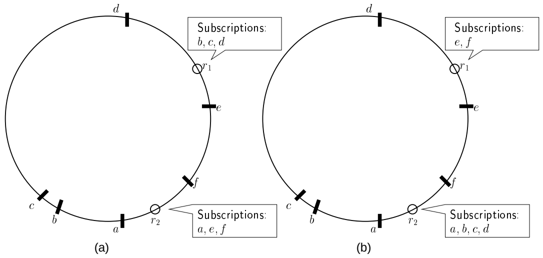


Figure 1: Illustration of RndRP and PredRP. r_1 and r_2 are two RP nodes. a, b, c, d, e, f are subscribers. (a) RndRP (b) PredRP.

However, PredRP may cause uneven subscription distribution across the RP nodes of the scheme S. As shown in Figure 1(b), r_1 stores less subscriptions than r_2. To deal with the load balancing issue, we propose a scheme, called *one-hop subscription push* (*one-hop push* for short) which is discussed in next subsection.

When a subscriber wishes to unregister his/her subscriptions previously installed in the system (e.g., due to changed interests), the subscriber can request the corresponding RP nodes to remove his/her subscriptions. Otherwise, we may associate each subscription with a TTL (time-to-live). However, the subscriber has to refresh his/her subscriptions periodically if he/she wants to continue receiving future events relevant to the subscriptions.

4.3 Subscription Management

Recall that Chord nodes consult their *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 unique sid. The routing paths from a RP node r to all these sids (or subscribers) form a tree (formed by DHT overlay links) rooted at the RP node r, say $EmdTree_r$ (an embedded tree rooted at r) [1]. As will be discussed in event delivery algorithm, the events will be disseminated along this tree from the RP node to the subscribers. Note that this tree is formed by the underlying DHT links, thereby imposing no additional construction or maintenance cost.

How does a RP node r manage the subscriptions installed by the subscribers? As outlined in Algorithm 3, r manages the subscriptions in a manner that a subscription s is stored according to the entry of a neighbor node (including *successor* nodes and *finger table* nodes) *whose node ID is equal to or most immediately precedes s's sid* [2]. Put in another way, we store the subscription

[1] Other DHTs such as Pastry and Tapestry have similar embedded trees as well.

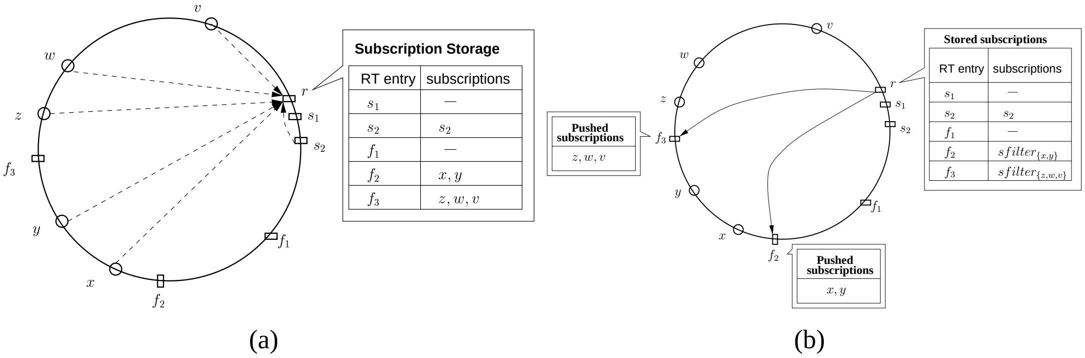
[2] This manner of subscription management is based on the observation that when routing a message from the RP node r to node s, r will first forward the message to its neighbor node whose ID is equal to or most immediately precedes s's ID.

s in the entry of a neighbor node which is the ancestor node of the corresponding subscriber in the embedded tree $EmdTree_r$. Note that this does not necessarily suggest that we put the data structure of subscriptions into r's routing table. We may just keep the metadata of the subscriptions into the entry of its routing table. However, how to associate subscriptions with routing table entries is not focus of the paper.

Algorithm 3 $manage\_subscriptions$(Subscription s)

```
Require: vector<Subscription> $store[1..k]$ //stores subscriptions in the RP node according to the entry of $k$ neighbor nodes
find the neighbor node $n_j$ whose ID is equal to or most immediately precedes $s$'s $sid$
$store[j].push\_back(s)$ //store $s$ into neighbor $n_j$'s entry
```

Figure 2(a) illustrates a RP node r's subscription management (for simplicity of presentation, the subscriptions of subscribers s_2, x, y, z, v and w in r are represented by their sids). Subscriber s_2's subscription is stored corresponding to the entry of r's successor node s_2. The subscriptions of subscribers x and y are stored corresponding to the entry of r's finger table node f_2, since f_2 is the ancestor node in the embedded tree $EmdTree_r$ (i.e., the routing path from r to x and the routing path from r to y will go through f_2). The subscriptions of subscribers z, w and v are stored corresponding to the entry of r's finger table node f_3 since f_3 is the ancestor node of z, w and v. As will be shown in Section 4.5, this novel subscription management can allow a RP node to deliver events by aggregating messages along its DHT links (i.e., links to its successor nodes and finger table nodes), thereby reducing the number of messages across the system.



Subscription Storage

RT entry	subscriptions
s_1	—
s_2	s_2
f_1	—
f_2	x, y
f_3	z, w, v

Stored subscriptions

RT entry	subscriptions
s_1	—
s_2	s_2
f_1	—
f_2	$sfilter_{\{x,y\}}$
f_3	$sfilter_{\{z,w,v\}}$

Figure 2: r is a RP 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. (a) Illustration of RP node r's subscription management. s_2, x, y, z, w, and v are subscribers whose subscriptions are stored in r. (b) One-hop subscription push. For simplicity, we use s_2, x, y, z, w and v to represent their corresponding subscriptions. $sfilter$ represents the summary filter.

One-hop subscription push. The basic idea is that a RP node r pushes the subscriptions corresponding to r's finger node f to f. r then uses a *summary filter* [3] to represent the subscriptions

[3] A summary filter covers the subscriptions pushed away by exploiting covering relationships between subscriptions [28].

pushed away. Upon an event e, r matches the event with the summary filter. If it is a match, r delivers e to the corresponding finger node which in turn serves as a RP node *agent* for those subscriptions pushed by r and starts delivering e to the matched subscribers. Figure 2(b) illustrates one-hop subscription push. r pushes subscriptions $\{x, y\}$ and $\{z, w, v\}$ to r's finger nodes f_2 and f_3, respectively.

One-hop push serves two main purposes. The first purpose is to allow a RP node to move part of its load (including subscriptions and event matching load) to some (or all) of its finger table nodes for load balance. For example, if a RP node r is overloaded by subscriptions, it finds a finger table node f is underloaded or willing to take some subscriptions through the load status piggybacked in the finger table maintenance messages which are sent periodically by the underlying DHT routing table maintenance process. Then, r could push those subscriptions corresponding to the entry of f to f. Note that the subscriptions to be pushed could also be piggybacked onto the routing table maintenance messages to reduce the number of messages involved. The second purpose is to reduce the message size from a RP node to its finger table nodes (at this point, no subscriber ID list is carried in the messages) during event delivery (see Section 4.5) and thus the bandwidth cost (see Section 6.2).

One-hop push works even if a finger table node f (which hosts the subscriptions pushed from a RP node r) leaves the system. This is because another finger table (say, g) will be picked by the routing table maintenance process performed periodically. If the g happens to be f's successor node, r does not need to push the subscriptions to g due to P2P data replication mechanism (i.e., g may already have one copy of the subscriptions). Otherwise, r needs to push the subscriptions to g or have g go to f's successor nodes for the copy (we assume a data replication mechanism for subscriptions). If there is a subscription s whose subscriber ID becomes not to follow g's ID anymore, the subscription s needs to be pushed back to r, which will assign s to the corresponding entry of the routing table.

4.4 Event Publication and Matching

When a node wishes to publish an event, it first directs the event to *all* the RP nodes corresponding to the scheme S. The RP nodes are responsible for matching the event to the subscriptions and starting delivering the event to the matched subscribers. Algorithm 4 outlines the process of event publication. It is worth pointing out that the event may be sent to the RP nodes either through the underlying Chord routing protocol, or through the direct point-to-point communication if the event publisher node has already cached the IP addresses of the RP nodes. The direct point-to-point communication between the publisher node and the RP nodes is expected to achieve better performance compared to the Chord routing protocol. However, if the number of the RP nodes is large (proportional to the number of attributes in S), either the point-to-point communication model may be inappropriate and impractical, or resorting to the Chord routing protocol may be inefficient (in terms of bandwidth). We may need to use a more efficient mechanism to publish the event to the RP nodes, e.g., multicast techniques. More discussion of this will be presented in Section 4.7.

Algorithm 4 $publish\_event$(Event e)

```
for each $A_i \in S$ do
    $k_i = h(A_i)$
    send $e$ to a RP node which is an immediate successor node of $k_i$
end for
```

Upon receiving an event e, each RP node needs to match e with the subscriptions stored on it. Algorithm 5 outlines the matching process which returns the matched subscriber ID lists with respect to the entry of the RP node's k neighbor nodes. Note that Algorithm 5 is a linear subscription matching algorithm with respect to the number of subscriptions. The matching from an event to a large number of subscriptions therefore could be very inefficient and the RP node may be overburdened by the matching load. To overcome this linear matching inefficiency, we could adopt *sublinear* matching algorithms based on building a subscription tree that collapses similar subscriptions [29]. However, how to optimize the matching algorithm is not focus of this paper. Algorithm 5 is primarily for illustration purpose.

Algorithm 5 $match\_subscriptions$(Event e)

```
Require: vector<Subscription> $store[1..k]$ //stores subscriptions in the RP node according to the entry of $k$ neighbor nodes
Require: $is\_match(e, p)$ returns TRUE if $e$ satisfies $p$, FALSE otherwise
Ensure: vector<ID> $matched\_set[1..k]$ //the matched subscribers' IDs to be returned
for each neighbor node $n_i$ do
    for each subscriptions $s_j = (sid_j, p_j) \in store[i]$ do
        if $is\_matched(e, p_j)$ then
            $matched\_set[i].push\_back(sid_j)$ //add the matched $sid$
        end if
    end for
end for
return $matched\_set$
```

To reduce the matching load on a RP node, one-hop subscription push allows the RP node to split and distribute its matching load to its $O(\log N)$ neighbor nodes. Moreover, with *attribute partitioning* (see Section 4.6) Ferry can distribute the matching load over more RP nodes.

4.5 Event Delivery

After event matching, how does a RP node r deliver the event to its subscribers by exploiting the embedded tree $EmdTree_r$? The basic idea behind Ferry's event delivery algorithm is that all the event delivery messages to those subscribers who share common ancestor nodes on the tree $EmdTree_r$ are *aggregated* into one single message along the path from the root node r to their lowest common ancestor node, thereby minimizing the number of messages. Algorithm 6 and Algorithm 7 outline the event delivery algorithm. The event delivery starts from the RP node r which sends out an event delivery message *carrying a corresponding subscriber ID list* (e.g., $matched\_set[i]$ in Algorithm 5) along its neighbor links (as shown in Figure 3). Upon receiving the message, each neighbor node (e.g., node s_2, f_2, or f_3 in Figure 3) executes $route\_message()$: if there is a subscriber ID matching its own ID, then it delivers the event to its local applications/users; it also partitions the remaining subscriber IDs (if any) in the message according to its own neighbor nodes (i.e., for each subscriber ID, 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. Note that all RP nodes of the scheme S will perform this event delivery operation in parallel.

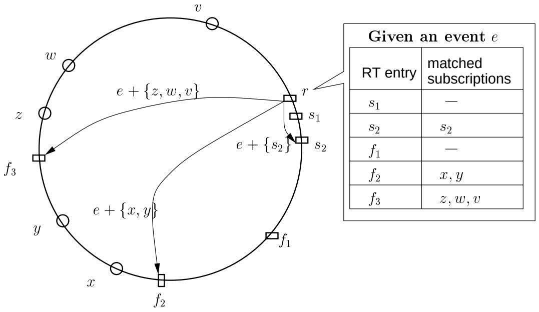
Algorithm 6 $deliver\_event$(Event e, vector<ID> $matched\_set[1..k]$)

```
for $i = 1$ to $k$ do
    if $matched\_set[i]$ is not empty then
        Message $M \leftarrow e + match\_set[i]$ //+ is a concatenation operator
        send $M$ to the neighbor node $n_i$, which then calls $route\_message(M)$ upon receiving $M$
    end if
end for
```

Algorithm 7 $route\_message$(Message M)

```
vector<ID> $matched\_set[1..k]$
Event $e \leftarrow$ extract the event from $M$
vector<ID> $list \leftarrow$ extract the list of subscriber IDs from $M$
for each $sid_i \in list$ do
    if $sid_i ==$ this node's ID then
        deliver $e$ to its local applications or users
    else
        find the neighbor node $n_j$ whose node ID is equal to or most immediately precedes $sid_i$
        $matched\_set[j].push\_back(sid_i)$
    end if
end for
if $matched\_set$ is not empty then
    $deliver\_event(e, matched\_set)$
end if
```

This event delivery algorithm is essentially a *recursive* process where each node along the dissemination paths of $EmdTree_r$ performs $deliver\_event()$ until the event reaches all subscribers. *No* event matching operation is performed along the dissemination path except the RP node, due to the subscriber ID list contained in the message.



RT entry	matched subscriptions
s_1	—
s_2	s_2
f_1	—
f_2	x, y
f_3	z, w, v

Figure 3: Illustration of the delivery of an event e from a RP node r. s_1 and s_2 are r's successors. f_1, f_2, and f_3 are members of r's finger table. s_2, x, y, z, w, and v are subscribers matching the event e.

4.6 Load Balancing

Note that for a pub/sub scheme S with n attributes, the maximum number of RP nodes is n. All subscriptions of S will be stored on and all events will be routed to these RP nodes. If the application/service corresponding to S is very popular, the subscriptions and events may overload the RP nodes. Therefore, Ferry takes three steps to tackle the load balancing issue.

First, load balancing is based on the randomness guarantee of the consistent hash function used for generating RP nodes for pub/sub services. Note that Ferry serves as a platform to host multiple pub/sub services. Hence, in the presence of multiple pub/sub services running on top of Ferry, Ferry distributes the onus of event publication, event matching, and subscriptions across all nodes: each node could be a RP node for some applications/services, and serve as the intermediate node to route events for other RP nodes. Second, one-hop push is used to reduce the load of a RP node by moving part of its subscriptions and event matching load to its neighbor nodes. However, one-hop push may not work if a RP node's neighbor nodes are all overloaded or unwilling to take the load.

Third, Ferry adopts *attribute partitioning* [28] to deal with the load balancing issue, by distributing load over more RP nodes. For example, consider that a scheme S has an attribute $temperature$ and the value range for $temperature$ is $[0, 100]$. Without partitioning, there is only one RP node. If we partition $temperature$ into several continuous ranges, $[0, 25]$, $(25, 50]$, $(50, 75]$, and $(75, 100]$, we may create 4 RP nodes by hashing the attribute name with a range. Thus, the load can be split by more RP nodes. Note that with attribute partitioning, we need to adapt the RndRP, PredRP, and event publication algorithm accordingly. However, the adaptation is very straightforward and simple. Algorithm 8 illustrates PredRP's subscription installation with attribute partitioning. Lines 1-10 produce a set of IDs which are derived from either the attribute name or the attribute name with a partition if the corresponding attribute has been partitioned. Note that the set rp_set is cached in memory for reuse to save computation cost. We put lines 1-10 in the algorithm just for illustration purpose.

Algorithm 8 $PredRP$(Subscription s)

```
 1: $rp\_set \leftarrow \emptyset$
 2: for each attribute $A_i \in S$ do
 3:   if $A_i$ with attribute partitioning then
 4:     for each partition $[L_k, H_k] \in A_i$ do
 5:       $rp\_set = rp\_set \cup \{h(A_i + L_k + H_k)\}$ //+ is a concatenation operator
 6:     end for
 7:   else
 8:     $rp\_set = rp\_set \cup \{h(A_i)\}$
 9:   end if
10: end for
11: $k \leftarrow$ choose a $h_i \in rp\_set$ such that $h_i$ is equal to or most immediately precedes $s$'s subscriber ID
12: store $s$ in a RP node which is an immediate successor node of $k$
```

4.7 Scaling to Number of Event Attributes

In event publication, an event can be either directly sent or routed (by Chord routing protocol) to the RP nodes of a scheme S. If the number of the RP nodes (which is determined by the number of event attributes of the scheme S and also attribute partitioning if applied) is small, the event publisher node can directly send the event to the RP nodes (by caching the IP addresses of the RP nodes) for better performance. However, if the number of RP nodes is very large, say, tens or even hundreds, using point-to-point communication would be impractical and inefficient. This is actually a problem of *how to efficiently deliver an event from the publisher node to a large number of RP nodes*. Fortunately, Ferry's novel event delivery mechanism has already provided an elegant solution to this problem, by *envisioning the RP nodes as the subscribers of the event publisher node*. Hence, unlike Meghdoot [14] which is built on top of CAN with $2n$ dimensions (n is the number of attributes), Ferry can support a pub/sub scheme with a large number of attributes.

4.8 Discussion

To the best of our knowledge, Ferry is the *first* solution that *extensively yet smartly* exploits the DHT overlay links to manage subscriptions and disseminate events for content-based pub/sub services. Its load-balancing technique, one-hop subscription push, also makes wise use of the DHT links. By exploiting DHT links in its design, Ferry have numerous advantages: (1) The fault-tolerance and self-organizing nature of DHT links makes Ferry resilient to node failures. (2) Ferry does not construct or maintain multicast trees for event delivery, and it is virtually maintenance-free. (3) Grouping subscriptions along DHT links in subscription management at the RP nodes facilitates message aggregation during event delivery, thus minimizing the number of messages across the system. (4) The PNS property of DHT links naturally enables proximity-aware event delivery along the DHT links, yielding good delivery performance. (5) The DHT routing table maintenance messages could be piggybacked onto the event delivery messages to reduce the DHT maintenance cost that is nontrivial in terms of bandwidth.

Ferry's event delivery algorithm is essentially a *match-first* approach: an event is first matched against all subscriptions in a RP node, generating subscriber ID lists each of which corresponds to a neighbor node (or DHT link). With the list contained in the event message, the event is routed to all subscribers on this list. The subscriber ID list may be undesirably long in a large network with thousands of subscribers and it may be infeasible to transmit and process large messages containing a long subscribe ID list throughout the network. However, note that the subscriber ID list contained in each event delivery message sent from a RP node is the split of the matched subscribers on the RP node among its $O(\log N)$ neighbor nodes. Thus, the length of list contained in each event delivery message from the RP node to its neighbor nodes would be reduced significantly, i.e., by an expected factor of $O(\log N)$. Moreover, the subscriber ID list carried in each event delivery message is split *recursively* by $O(\log N)$ nodes at each step along the event dissemination path on the embedded tree. As a result, the size of the subscriber ID list is expected to be reduced by a factor of $O(\log N)$ at each step along the dissemination path. In addition, our two load-balancing techniques further make this a lesser issue. First, with attribute partitioning, Ferry can significantly reduce the load on a RP node by distributing load over more RP nodes, resulting in reduced size of the subscriber ID list to be contained in each event delivery message. Second, one-hop subscription push eliminates the subscriber ID list from the RP node to its neighbor nodes. Only the event delivery messages from the RP node's neighbor nodes will contain a subscriber ID list. As such, one-hop push can reduce the size of the subscriber ID list contained in each event delivery message by a factor of $O(\log^2 N)$. This is because, with one-hop push, the matched subscriber IDs will be split first among a RP node's $O(\log N)$ neighbor nodes and then each such a neighbor node's $O(\log N)$ neighbor nodes.

4.9 Adaptability to Other DHTs

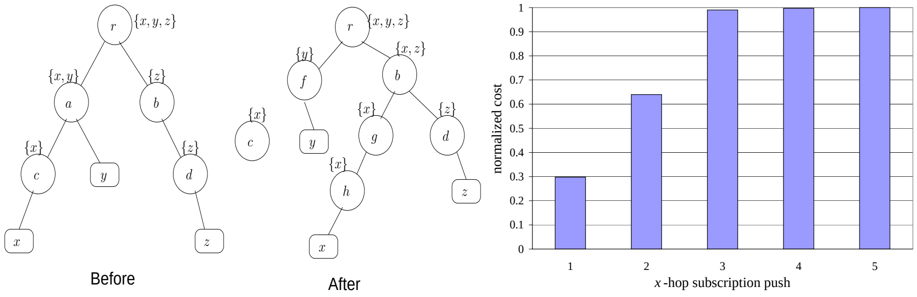
Although we base Ferry's design on Chord, the proposed techniques in Ferry are applicable or easily adaptable to other DHTs such as Pastry, Tapestry and CAN. As discussed above, Ferry's subscription installation and management, event delivery and load balancing technique one-hop subscription push, all extensively yet wisely exploit the embedded tree structures (or DHT links) in the underlying DHT. In spite of different DHT geometries such as tree (e.g., Pastry and Tapestry), ring (e.g., Chord) and hypercube (e.g., CAN), all these DHTs have embedded trees formed by the DHT links. Thus, we believe that Ferry can be easily adapted to tree-like and hypercube-like DHTs. We here take Pastry as an example and demonstrate how to extend Ferry into Pastry.

Pastry is essentially a prefix-based routing protocol: routing is achieved by successively "correcting" the highest order bit on which the forwarding node differs from the destination node, effectively increasing the length of the longest prefix match by one at each hop. In subscription installation, PredRP stores a subscription into a RP node which shares the longest common prefix with the subscriber node ID (if two or more RP nodes have same longest length of matching prefix with the subscriber node ID, PredRP stores the subscription into the RP node whose ID is numerically closest to the subscriber ID). The intuition is that, in a tree geometry, node IDs constitute the leaf nodes in a binary tree of depth $\log N$, and the "distance" between any two nodes is the height of their smallest common subtree. Thus, in tree-like DHTs such as Pastry, PredRP is able to minimize overlay hops of event delivery and subscription installation, as each subscription is installed on the "closest" RP node. Furthermore, when events are disseminated from the RP nodes, we can avoid sending redundant messages across the Pastry overlay, thereby improving event delivery performance. Event message delivery is constrained within the smallest common subtrees shared by the subscribers and the RP nodes.

In subscription management, each RP node groups subscriptions stored on it to its neighbor nodes (including routing table nodes and leafset nodes) which shares the longest common prefix with their corresponding subscriber IDs; if two or more neighbor nodes have same longest length of matching prefix with the subscriber node ID, the RP node puts the corresponding subscription(s) into the entry of the neighbor node whose ID is numerically closest to the subscriber ID. This manner of subscription grouping is based on the Pastry's prefix-based routing protocol. By having this subscription management, one-hop subscription push and event delivery in Ferry can be easily extended into Pastry and the extension is very straightforward.

5 Alternative Design

One alternative design we have considered for Ferry and finally abandoned is to install the subscriptions on *all the peer nodes along the dissemination path from the RP node to their subscribers* (as shown in Figure 4(a)). Upon receiving an event, each RP node computes which subset of its neighbors are to receive the event, i.e., it determines those DHT links along which it should transmit the event message. Each neighbor node receiving the message in turn forwards the event message downstream to the corresponding subset of its neighbors (through similar computation of the subset of neighbors). This process continues until the event message reaches the destinations. Note that there is *no* subscriber ID list carried in the event messages in this alternative design and all the nodes along the dissemination path have to do event matching.



(a) (b)

Figure 4: (a) Subscription installation and removal after node a's departure. r is a RP node, while x, y and z are subscriber nodes. For simplicity of presentation, x, y and z are also used to represent the corresponding subscriptions. (b) Normalized cost of subscription installation and removal as nodes continuously join and leave in a 1024-node network with node session times following a uniform distribution in which node session times are uniformly at random chosen between 6 minutes and nearly 2 hours with an average of 1 hour.

Compared to the Ferry's design, the alternative design however has the following limitations. First, it requires $O(\log N)$ times more storage space by installing subscriptions on the $O(\log N)$ nodes along the dissemination path and extra processing time for subscription matching during event delivery at every peer node along the path. Second, both subscribing and un-subscribing have to involve $O(\log N)$ more nodes and messages. A high rate of subscribing/un-subscribing would result in heavy traffic of subscribing/un-subscribing across the system. Moreover, if the subscriptions are associated with a TTL, the subscription refreshing traffic would be heavy.

Third, node churn may cause frequent subscription installation and removal along the paths.

The left graph in Figure 4(a) shows subscriptions of x, y and z are installed on the peer nodes along the path from the RP node r to x, y and z, respectively. Assume $b.ID < a.ID < x.ID$ (x follows a and a follows b on the Chord ring clockwise). When r's neighbor node a departed from the system (as shown in the right graph of Figure 4(a)), r replaced it after repair with node f whose ID follows x.ID clockwise on the Chord ring [4]. As a result, x's subscription needs to be removed from node c (otherwise this may cause duplicated event messages delivered to x) and to be re-installed on all the peer nodes along the *new* dissemination path from r to x via b, g and h. Similarly, node joins could also cause subscription removal and installation along the paths. Finally, data replication, which is used to improve data availability in DHTs, would incur more overhead in storage and bandwidth dedicated to subscription installation and removal under node churn.

Due to the aforementioned disadvantages, we do not adopt this alternative design. However, it is worth pointing out that Ferry's design with one-hop subscription push strikes a balance between Ferry's design without one-hop push and this alternative design. One-hop push, one the one hand, aims to reduce the load on RP nodes and the event delivery message size; on the other hand, it tries to avoid the drawbacks of the alternative design. If Ferry pushes the subscriptions on RP nodes $O(\log N)$ hops away along their dissemination paths, then Ferry will degenerate into this alternative design. Figure 4(b) shows the cost (number of subscription installation and removal) incurred by x-hop subscription push normalized to this alternative design in the face of node churn. As nodes join and leave continuously, one-hop subscription push introduces the least cost, only about 0.3 of the cost incurred by the alternative design (4-hop subscription push represents the alternative design in the 1024-node network). The results justify our design choice on Ferry.

6 Evaluation

6.1 Experimental Setup

We implemented Ferry on top of **p2psim** [5], a discrete-event packet level simulator. The **p2psim** implementation includes a detailed Chord simulator, and it does not simulate link transmission rate

[4] After each node failure/departure, the underlying DHT routing table maintenance process will detect the failure (e.g., node a) and fix it with new routing entry (e.g., node f). Thus, node churn will cause little overhead for event delivery algorithms which rely on the embedded tree structure inherent in DHTs.

[5] http://pdos.lcs.mit.edu/p2psim

or queuing delay [30]. The number of successor nodes for each Chord node is 16 and the finger table base is 2. We used the default values of the finger stabilization interval and successor stabilization interval accompanied with **p2psim**. Chord has a configuration named *proximity neighbor selection* (PNS) which allows each Chord node to choose physically close nodes as routing table entries to reduce lookup latency [27]. The simulated network used in our simulations consists of 1024 nodes with inter-node latencies derived from measuring the pairwise latencies of 1024 DNS servers on the Internet using King method [31]. The average round-trip time for the simulated network is 198 milliseconds. Unless otherwise specified, our experimental results presented in this paper are based on this network.

The simulations were initialized with one Chord node in the system. A new Chord 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 into the system to store the subscriptions. After subscription installation, the event publication was modeled as exponential distribution with an average inter-arrival time of 116 seconds.

The scheme S used in our experiments was proposed in Meghdoot [14], and defined as $S = \{[Date : string, 2/Jan/98, 31/Dec/02], [Symbol : string, “aaa”, “zzzzz”], [Open : float, 0, 500], [Close : float, 0, 500], [High : float, 0, 500], [Low : float, 0, 500], [Volume : integer, 0, 310000000]\}$. Specifically, *Symbol* is the stock name. *Open* and *Close* are the 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.

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 [14]. The number of stocks and subscriptions used in simulations were 100 and $10,000$ respectively by default, unless otherwise specified. The events were generated randomly from S and we used $100,000$ events in simulations.

We used a set of metrics to evaluate the performance and cost of Ferry: (1) *hops*: the average number of overlay hops taken by Ferry to deliver an event to all of its subscribers; (2) *latency*: the average time taken by Ferry to deliver an event to all of its subscribers; (3) *overhead*: it is defined as the ratio of the number of intermediate nodes involved during the delivery of an event to the number of subscribers for this event. The lower the overhead, the better performance of Ferry; (4) *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 header, 33 bytes for the event, and 4 bytes for each subscriber ID carried in the message.

It is worth pointing out that the results we present next do not include event publication and we primarily focused our experiments on Ferry's event delivery algorithm. Recall that, in event publication, an event can be either sent directly or routed to the RP nodes from the event publisher node. If the event is directly sent to the RP nodes, the average event publication latency would be average latency between nodes. If the event chooses to be routed to the RP nodes, as discussed in Section 4.7, it can use Ferry's event delivery algorithm to publish the event to the RP nodes by envisioning the RP nodes as the publisher node's subscribers. In this case, the performance and cost of event publication are similar to those of event delivery.

6.2 Experimental Results

In what follows, we first extensively investigate the performance of Ferry under various configurations and different network sizes. We then compare Ferry and Meghdoot in event delivery performance.

6.2.1 Performance under Various Configurations

We first present the results for Ferry's various configurations under $10,000$ subscriptions and $100,000$ events. The average number of subscribers per event is 25, about 2.4% of 1024 nodes. Table 1 shows the performance of Ferry with different configurations. Note that PredRP outperforms RndRP significantly, and PredRP+PNS performs the best. Compared to RndRP, PredRP can dramatically reduce hops, latency and overhead. This is because (1) the event delivery messages in PredRP traverse shorter ranges of the Chord ring space (the range an event may traverse in PredRP is bounded by the maximum Chord ring range between two contiguous RP nodes, while in RndRP the range may cover the whole Chord ring space), and (2) PredRP can avoid sending redundant messages across the Chord ring space.

Table 1. Comparison between different Ferry's configurations

scheme	hops	latency(ms)	bw_cost(Bytes/node)	overhead
RndRP	3.94	359.17	53.67	2.51
PredRP	2.64	235.28	52.41	1.20
RndRP+PNS	3.80	154.22	53.56	2.41
PredRP+PNS	2.57	144.34	52.16	1.18

Figure 5 plots the distribution of events for PredRP+PNS according to the range of hops, latency, bandwidth cost, and overhead. Note that 94.22% of the events are delivered to all subscribers within 2-3 hops. The average delivery hops for the events is 2.57. 59.2% of events are delivered with latency between 100-150ms, and the average delivery latency is 144.34ms. The average bandwidth cost and overhead are 52.16 Bytes/node and 1.18 per event, respectively.

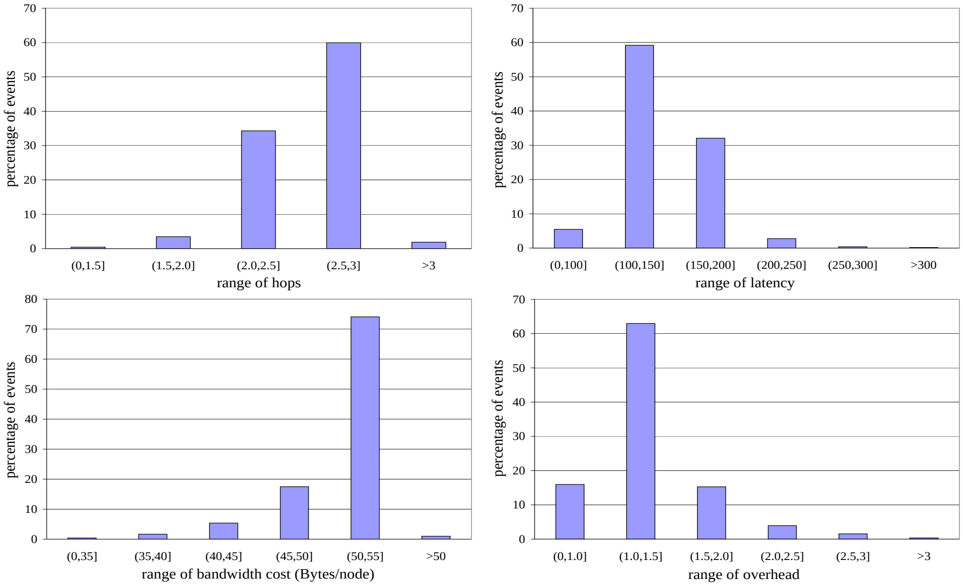


Figure 5: Distribution of events with respect to hops, latency, bandwidth cost, and overhead.

We present the event distribution with respect to the number of subscribers in Figure 6. Note that about 60% of the events have 20-40 subscribers. Figure 7 shows the overhead with respect to the number of subscribers. As the number of subscribers increases, the overhead decreases. This shows that Ferry can deliver events to a larger number of subscribers at lower overhead. This reduction in overhead mainly results from the synthesis of PredRP and message aggregation along the dissemination paths during event delivery.

Figure 8 shows the subscription distribution on 7 RP nodes for RndRP and PredRP. Note that RndRP evenly distributes subscriptions to the RP nodes while PredRP produces a skewed load distribution. This shows that it is very necessary for PredRP to apply one-hop subscription push for load balance. We also investigated the impact of one-hop push on bandwidth cost for RndRP+PNS

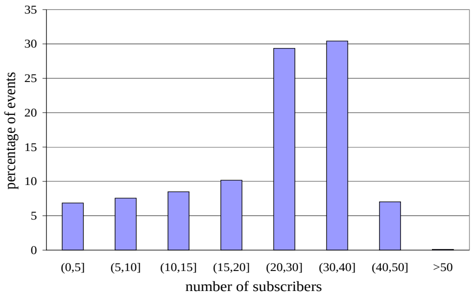


Figure 6: Event distribution by subscriber number range.

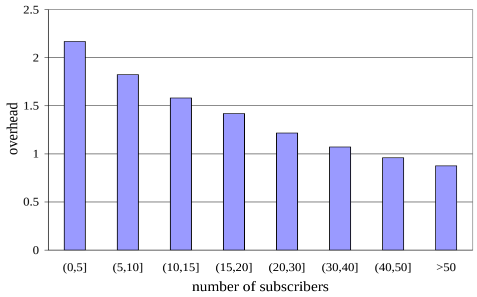


Figure 7: Overhead by subscriber number range.

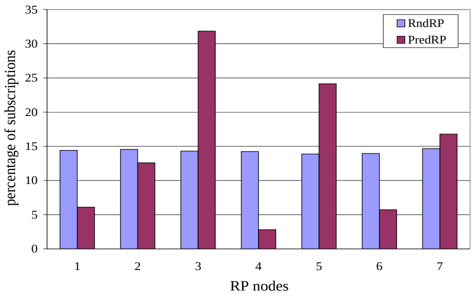


Figure 8: Subscription distribution in RndRP and PredRP.

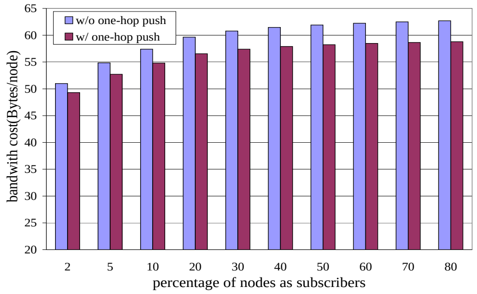


Figure 9: Bandwidth cost of PredRP+PNS with/without one-hop push.

and PredRP+PNS. The results show that one-hop push reduces the bandwidth cost *per event* for RndRP+PNS from 53.56 to 52.39 Bytes/node, and for PredRP+PNS from 52.16 to 50.34 Bytes/node. The bandwidth cost reduction results from the reduced message sizes from the RP nodes to their neighbor nodes (at this point, no subscriber ID list is carried in the messages). Note that the bandwidth cost reduction is per node/event, so a small reduction could result in huge reduction in aggregated bandwidth cost across the system.

Table 2. Results of PredRP+PNS for various percentages of nodes as subscribers per event

metric	2%	5%	10%	20%	30%	40%	50%	60%	70%	80%
hops	2.58	2.59	2.58	2.58	2.59	2.59	2.58	2.58	2.58	2.58
latency(ms)	143.46	143.97	143.88	143.59	143.94	143.96	144.19	143.95	143.95	143.96
bw_cost(Bytes/node)	50.99	54.85	57.40	59.65	60.78	61.46	61.91	62.24	62.49	62.69
overhead	1.36	0.86	0.53	0.28	0.18	0.12	0.08	0.05	0.03	0.02

The experimental results we present in the rest of the paper will focus on Ferry with the configuration of PredRP+PNS unless otherwise noted. To explore Ferry's performance with respect to the number of subscribers, we each time ran Ferry by delivering $100,000$ events each of which has a given number of subscribers randomly chosen from the system. Table 2 shows the results for various percentages of nodes as subscribers for each event. As the number of subscribers increases, the hops and latency almost keep constant at 2.6 and 144ms respectively, while the bandwidth cost

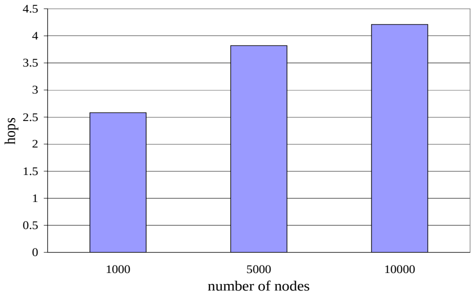


Figure 10: Overlay hops.

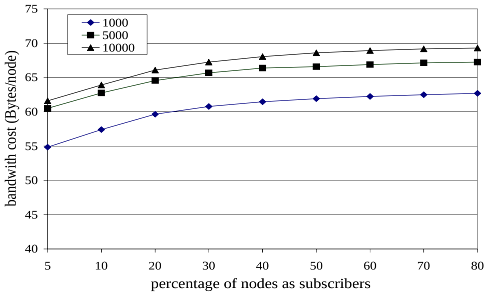


Figure 11: Bandwidth cost w/o one-hop push.

increases modestly. However, the overhead drops significantly. The results show that Ferry could deliver events to a large number of subscribers at very low overhead, involving only a small number of intermediate nodes by the synthesis of its message aggregation and the subscription installation algorithm PredRP. Hence, Ferry is very efficient in delivering events to a large number of subscribers. Figure 9 shows that one-hop push could effectively reduce the bandwidth cost due to the reduced message size from the RP nodes to their neighbor nodes. Note that the bandwidth reduction is per node/event, so a small reduction could result in huge reduction in aggregated bandwidth cost across the system.

6.2.2 Effect of Network Size

In this subsection we present the performance of Ferry in various network sizes of 1000, 5000, and 10000 nodes. The simulated networks of 5000 and 10000 were derived from the 1024-DNS server measurements. For a given network size, we ran simulations for various percentages (from 5% to 80%) of nodes randomly chosen as subscribers interested in an event and the total number of events is $100,000$ for a given percentage. We found the overhead is almost constant under various network sizes for a given percentage of node as subscribers; the overhead drops from 0.86 to 0.02 as the percentage of nodes as subscribers per event increases from 5% to 80%. Figure 10 shows the average number of hops taken by event delivery for network sizes of 1000, 5000, and 10000. Figure 11 shows the bandwidth cost (one-hop push was not used here). Note that as the network size increases, the bandwidth cost and overlay hops incurred by event delivery increase modestly. This shows that Ferry can scale to a large number of nodes. Moreover, with the number of subscribers interested in an event increasing (from 5% to 80%), the bandwidth cost increases slightly. This show that Ferry can scale to a large number of subscribers per event.

6.2.3 Impact of Attribute Partitioning

In this subsection, we present results of Ferry with attribute partitioning in a 5000-node system where the average RTT is 200ms. We first provide the results of attribute partitioning on the load of RP nodes, and then give the results of attribute partitioning on latency, overhead, overlay hops and bandwidth cost.

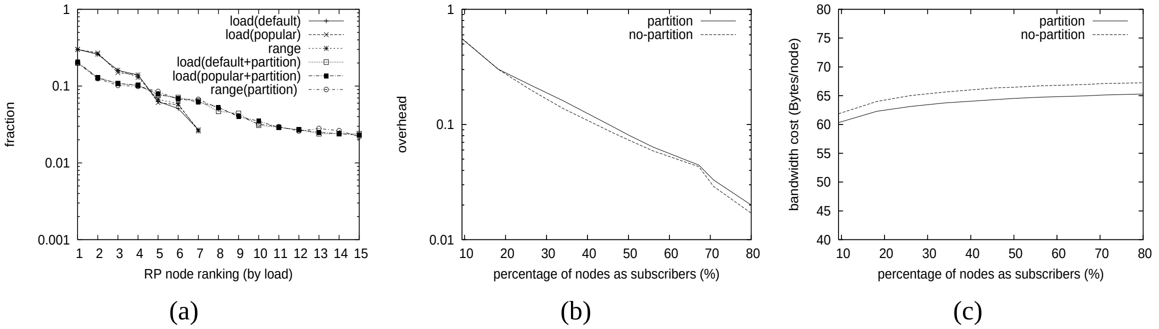


Figure 12: Results for PredRP+PNS where one-hop push is not applied. (a) Load and Chord ring range of RP nodes. (b) Overhead. (c) Bandwidth cost.

Figure 12(a) plots both *load* and *range* of RP nodes. The x-axis represents the ranking of RP nodes by their load and the y-axis is in log scale. *load* represents the fraction of subscriptions on a RP node while *range* means the fraction of Chord ring from which a RP node stores subscriptions [6]. *default* refers to the five subscription templates (see Section 6.1) we used to generate subscriptions. *popular* represents the revised five templates to reflect popular attributes, i.e., we assigned T_2 with probability 80% and other four templates each with probability 5%. Thus, Low is a very popular attribute. *partition* represents the case where we partitioned the dimension of each three attributes $Open$, Low and $High$ into four equal parts. With attribute partitioning, we produced 15 RP nodes (we found two hashing results mapped to one same RP node) and thereby could distribute the load into more RP nodes.

Two important observations can be drawn from Figure 12(a): (1) With attribute partitioning, we can reduce the load on RP nodes by distributing load over more RP nodes. Note that the load of top ranking RP nodes is reduced significantly due to attribute partitioning; (2) the popularity of an attribute and thus the skewed distribution of subscriptions have *no* impact on the load of RP nodes. This is because in PredRP, each RP node only stores the subscriptions from the Chord ring range

[6] One distinction should be made between the Chord ring range a RP node charges and the Chord ring range a Chord node is responsible for. The former range is the one from which the RP node stores the subscriptions while the latter is the Chord ring range between the Chord node and its predecessor (i.e., the Chord node is responsible for the IDs falling within this ring range).

between the RP node itself and the next immediately following RP node. Note that the *load* on a RP node is proportional to the *range* it charges (the *load* curve closely matches the *range* curve irrespective of attribute partitioning).

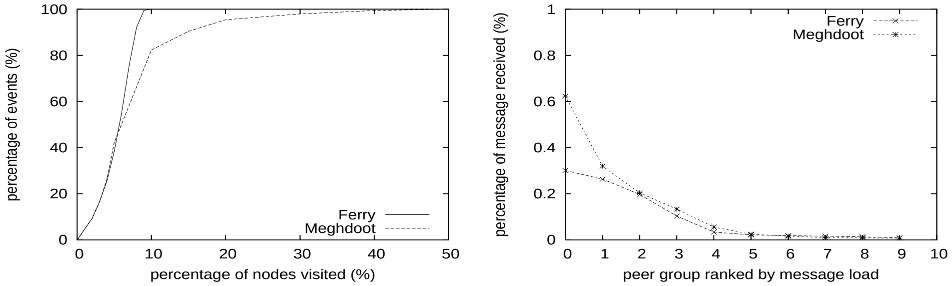
From the above observations, we can see that, in PredRP, a RP node takes load from the Chord ring range it charges, *irrespective of the popularity of attributes and thus skewed subscription distribution*. This feature is very desirable. However, unlike RndRP, PredRP introduces load imbalance among RP nodes due to various sizes of the Chord ring ranges the RP nodes charge. The various sizes of ring ranges result from the hashing process of producing RP nodes which are just *pseduo-uniform*. To reduce the load imbalance among RP nodes. We may take the following steps. First, with one-hop subscription push, the top ranking RP nodes may split some of their load into their neighbor nodes. Second, we may distribute load into more RP nodes by finer-grained partitioning of attributes. Finally, we can design algorithms which make RP nodes to charge equal-size range of Chord ring (for example, the i-th RP node for a scheme S can be the immediate successor node of $k = h(S) + \frac{i}{n+1} \cdot |R|$, where $h(S)$ is the content hash of S, n is the maximum number of RP nodes for S, and $|R|$ is the address space of the Chord ring), and then the load on RP nodes is expected to be equally distributed. However, this assumes that the number of subscriptions from the Chord ring is uniformly distributed. We leave this to our future work.

Next, we present the results of attribute partitioning with respect to the number of subscribers interested in an event. Several observations can be drawn from Figure 12(b) and (c): (1) As subscribers interested in an event increases, the bandwidth cost increases modestly while the overhead drops significantly; (2) attribute partitioning has little impact on overhead; (3) with attribute partitioning, Ferry could reduce the bandwidth cost significantly, due to the reduced message size which results from the reduced length of subscriber ID lists contained in the event delivery messages since more RP nodes split the subscriptions. We also found that, with attribute partitioning, Ferry could significantly decreases the latency by 9.75% (from 181.10ms to 163.45ms) and overlay hops by 12.84% (from 3.66 to 3.19) upon various numbers of subscribers per event. This is mainly because with attribute partitioning, event messages traverse shorter range of Chord ring space (since more RP nodes partitions the Chord ring space, as shown in Figure 12(a)).

6.2.4 Performance Comparison with Meghdoot

Ferry and Meghdoot have different design philosophies: Meghdoot builds a semantic CAN overlay for a content-based pub/sub scheme while Ferry serves as a platform to host multiple pub/sub services with unique schemes. As discussed in Section 4.6, Ferry takes different approaches to address the load balancing issue. It is unfair to compare the two systems on subscription distribution for a single pub/sub scheme. However, subscription installation in Ferry and Meghdoot takes $O(\log N)$ and $O(dN^{\frac{1}{d}})$ hops respectively, where N is the number of nodes and d is the dimensionality of the CAN overlay space. If $d = (\log_2 N)/2$, Ferry and Meghdoot achieve same subscription installation performance. Note that $d = 2n$ in Meghdoot where n is number of attributes in a pub/sub scheme.

It is meaningful to compare Ferry and Meghdoot in terms of event delivery performance because event delivery is critical to a pub/sub system. We use 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.



(a)

(b)

Figure 13: Comparisons in a 1024-node system. (a) CDF of event distribution with respect to the number of nodes visited. (b) Distribution of event delivery load by peer group.

Figure 13(a) 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. Ferry shows better performance than Meghdoot in that all event deliveries end up with visiting at most 10% nodes. The is mainly due to the synthesis of Ferry's PredRP algorithm and 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 13(b) shows the average load on each group. The load distribution among peers is more balanced in Ferry than Meghdoot. For example, the maximum load on a node in Ferry is about 0.3% of the total messages, which is very good. This shows that Ferry is able to fairly distribute event delivery load among the nodes in the system.

7 Conclusions and Future Work

Ferry is the first design that extensively yet wisely exploits the underlying DHT overlay structures to build an efficient and scalable platform for content-based pub/sub services. Its novel subscription installation and management, event delivery and load balancing technique one-hop subscription push all make wise use of the DHT links. Via detailed simulations, we show that Ferry can deliver events to various numbers of subscribers under different network sizes efficiently in terms of bandwidth and overhead and timely in terms of overlay hops and latency. Moreover, Ferry can support a content-based pub/sub scheme with a large number of event attributes. Our preliminary results show that Ferry has better performance than Meghdoot in event delivery. However, we confess that the comparison between the two systems is by no means complete.

This paper constitutes an initial step to build an efficient and scalable platform for content-based pub/sub. A number of issues need to be explored in our next steps. For instance, we will investigate the reduction of the DHT maintenance cost in terms of bandwidth by piggybacking the DHT link maintenance messages onto the event delivery messages. Another problem we will study is how cooperative peer nodes have to be in Ferry. For example, if we are disseminating stock data, there is inherent interest for an intermediate node to delay delivery until it can take advantage of the data first. Thus, potential applications of Ferry for large-scale pub/sub have to consider this issue, i.e., to provide incentives for nodes to cooperate in event delivery. We also plan to perform a thorough comparison between Ferry and existing DHT-based content-based pub/sub systems such as Meghdoot.

References

[1] 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 *Proceedings of the 2001 ACM SIGMOD*, vol. 30, (Santa Barbara,CA), pp. 115–126, 2001.

[2] B. Segall and D. Arnold, "Elvin has left the building: A publish/subscribe notification service with quenching," in *Proceedings of AUUG*, Sept. 1997.

[3] 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 *Proceedings of the 19th IEEE ICDCS*, pp. 262–272, 1999.

[4] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Design and evaluation of a wide-area event notification service," *ACM Transactions on Computer Systems*, vol. 19, no. 3, pp. 332–383, 2001.

[5] P. Triantafillou and A. Economides, "Subscription summarization: A new paradigm for efficient publish/subscribe systems," in *Proceedings of the 24th IEEE ICDCS*, 2004.

[6] A. Carzaniga and A. L. Wolf, "Forwarding in a content-based network," in *Proceedings of ACM SIGCOMM*, (Karlsruhe, Germany), pp. 163–174, Aug. 2003.

[7] A. Carzaniga, M. J. Rutherford, and A. L. Wolf, "A routing scheme for content-based networking," in *Proceedings of IEEE INFOCOM*, (Hongkong, China), Mar. 2004.

[8] F. Cao and J. P. Singh, "MEDYM: An architecture for content-based publish-subscribe networks," in *Proceedings of ACM SIGCOMM*, (Portland, OG), Aug. 2004.

[9] 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 *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems(ICDCS)*, (Austin, TX), pp. 262–272, May 1999.

[10] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of ACM SIGCOMM*, (San Diego, CA), pp. 149–160, Aug. 2001.

[11] A. Rowstron and P. Druschel, "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, Nov. 2001.

[12] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph, "Tapestry: An infrastructure for fault-tolerance wide-area location and routing," Tech. Rep. UCB/CSD-01-1141, Computer Science Division, University of California, Berkeley, Apr. 2001.

[13] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *Proceedings of ACM SIGCOMM*, (San Diego, CA), pp. 161–172, Aug. 2001.

[14] A. Gupta, O. D. Sahin, D. Agrawal, and A. E. Abbadi, "Meghdoot: Content-based publish/subscribe over P2P networks," in *ACM/IFIP/USENIX 5th International Middleware Conference*, (Toronto, Ontario, Canada), Oct. 2004.

[15] Y. Choi, K. Park, and D. Park, "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 2004.

[16] P. Triantafillou and I. Aekaterinidis, "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 2004.

[17] W. W. Terpstra, S. Behnel, L. Fiege, A. Zeidler, and A. P. Buchmann, "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 2003.

[18] D. Tam, R. Azimi, and H.-A. Jacobsen, “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), Sept. 2003.

[19] G. Perng, C. Wang, and M. K. Reiter, “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 2004.

[20] P. R. Pietzuch and J. Bacon, “Peer-to-peer overlay broker networks in an event-based middleware.,” in *Proceedings of the Second International Workshop on Distributed Event-Based Systems (DEBS)*, (San Diego, CA), June 2003.

[21] A. I. T. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel, “SCRIBE: The design of a large-scale event notification infrastructure,” in *Proceedings of the 3rd International Networked Group Communication*, pp. 30–43, 2001.

[22] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. Kubiatowicz, “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)*, June 2001.

[23] K. P. Birman, “The process group approach to reliable distributed computing,” *Communications of the ACM*, vol. 36, pp. 36–53, Dec. 1993.

[24] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen, “The information bus: an architecture for extensible distributed systems,” in *Proceedings of the fourteenth ACM SOSP*, (Asheville, NC), pp. 58–68, Dec. 1993.

[25] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh, “Splitstream: High-bandwidth multicast in cooperative environments,” in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, (Bolton Landing, NY), Oct. 2003.

[26] A. Bharambe, S. Rao, V. Padmanabhan, S. Seshan, and H. Zhang, “The impact of heterogeneous bandwidth constraints on dht-based multicast protocols,” in *Proceedings of the 4th International Workshop on Peer-to-Peer Systems(IPTPS)*, (Ithaca, NY), Feb. 2005.

[27] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris, “Designing a DHT for low latency and high throughput,” in *Proceeding of the First Symposium on Networked Systems Design and Implementation (NSDI)*, (San Francisco, CA), pp. 85–98, Mar. 2004.

[28] Y.-M. Wang, L. Qiu, D. Achlioptas, G. Das, P. Larson, and H. J. Wang, “Subscription partitioning and routing in content-based publish/subscribe systems,” in *Proceedings of the 16th International Symposium on Distributed Computing (DISC)*, (Toulouse, France), Oct. 2002.

[29] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra, “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 1999.

[30] J. Li, J. Stribling, T. M. G. andRobert Morris, and M. F. Kaashoek, “Comparing the performance of distributed hash tables under churn,” in *Proceedings of The 3rd International Workshop on Peer-to-Peer Systems (IPTPS)*, (San Diego, CA), Mar. 2004.

[31] K. P. Gummadi, S. Saroiu, and S. D. Gribble, “King: Estimating latency between arbitrary internet end hosts,” in *Proceedings of the 2002 SIGCOMM Internet Measurement Workshop*, (Marseille, France), Nov. 2002.