Ferry: An Architecture for Content-Based Publish/Subscribe Services on P2P Networks

Yingwu Zhu
Department of ECECS
University of Cincinnati
zhuy@ececs.uc.edu

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

Abstract

Leveraging DHTs (distributed hash table), we propose Ferry, an architecture for content-based publish/subscribe services. With its novel design in subscription installation, subscription management and event delivery algorithm, Ferry can serve as a scalable platform to host any and many content-based publish/subscribe services: any publish/subscribe service with a unique scheme can run on top of Ferry, and many publish/subscribe services can run together on top of Ferry. For each of the publish/subscribe services running on top of Ferry, Ferry does not need to maintain or dynamically generate any dissemination tree. Instead, it exploits the embedded trees in DHTs such as Chord to deliver events, thereby imposing little overhead. By delivering events along DHT links, Ferry has two main advantages: (1) it eliminates the cost in construction and maintenance of the dissemination trees; (2) it allows further optimization, i.e., the DHT link maintenance messages could be piggybacked onto the event delivery messages to reduce the maintenance cost which is inherent and nontrivial in DHTs. Moreover, Ferry can support a publish/subscribe scheme with a very large number of event attributes.

1. Introduction

Content-based publish/subscribe (pub/sub) is a powerful paradigm for information dissemination from *publishers* to *subscribers* in a large-scale distributed network. In a content-based pub/sub system, subscribers register their interests in future *events* through *subscriptions*. Upon receiving an event (published by a publisher), the system matches the event to the subscriptions which serve as *filters* and deliver the event to the matched subscribers. The major advantage of content-based pub/sub is its high expressiveness in subscriptions, i.e., a subscription is expressed by specifying a set of *predicates* over event attributes [9].

As DHT-based peer-to-peer (P2P) systems [17, 15, 23, 14] attract more and more interests from the research community due to their scalability, fault-tolerance and self-organization, many content-based pub/sub systems [11, 7, 20, 19, 18, 12, 13] have been recently built on top of these DHTs. In such systems, peers cooperate in storing subscriptions and routing events to their subscribers in a fully distributed manner. A big challenge facing a P2P-based pub/sub system is to design a light-weight, efficient, and timely event delivery algorithm. Put in another way, the pub/sub system should impose small overhead on the underlying DHT, and the event delivery should be efficient in terms of bandwidth cost and timely in terms of user-perceived latency.

To this end, we propose an architecture for content-based pub/sub services, called *Ferry*, built on top of Chord [17]. As a platform, Ferry can host *any* and *many* pub/sub services. This is twofold: (1) any pub/sub service with a unique scheme can run on top of Ferry; (2) many pub/sub services can coexist on top of Ferry. For each pub/sub service running on top of Ferry, Ferry does not need to maintain or dynamically generate any dissemination tree to deliver events. Leveraging the embedded trees (formed by DHT links) in the DHT, Ferry aggregates and delivers event messages along DHT links, thereby imposing little overhead. Exploiting DHT links has two major advantages. First, it eliminates the cost in construction and maintenance of dissemination trees used for event delivery. Second, it 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 DHTs. To deal with the load balancing issue, Ferry takes three steps. First, it relies on the uniformity of the consistent hash function used in Chord to distribute subscriptions and events across nodes. Second, it proposes a scheme, called *one-hop subscription push* to balance the subscription distribution among neighbor nodes. Finally, it adopts *attribute partitioning* [22] to further improve load

balance.

We have built Ferry on top of p2psim [1], a discrete-event packet level simulator. Via detailed simulations, we evaluated Ferry extensively in terms of overlay hops, latency, overhead, and bandwidth cost. The experimental results show that Ferry can deliver events to a large number of subscribers with very small overhead and latency. Moreover, Ferry can support a pub/sub scheme with a very large number of event attributes.

The rest of the paper is structured as follows. Section 2 provides a survey of related work. We present the design of Ferry in Section 3. Section 4 presents experimental setup and results. We conclude this paper in Section 5.

2. Related Work

Due to space constraints, we here just present the most related work. Many distributed content-based pub/sub systems [2, 4, 21, 6, 5, 3] have been proposed by using routing trees to deliver events to the subscribers based on multicast techniques. Among these systems, Ferry is most similar to MEDYM [3]. 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 does not need to dynamically generate a dissemination tree on demand and it instead exploits the embedded trees inherent in a DHT to deliver events, thereby imposing little overhead.

DHTs such as Chord [17], Pastry [15], Tapestry [23], and CAN [14] offer an attractive platform to build content-based pub/sub systems due to their scalability, load balance, fault-tolerance, and self-organization. Many attempts have been made in designing a P2P-based pub/sub system [19, 20, 18, 12, 7, 13, 16, 25, 11]. Tam et al. [18] propose a content-based pub/sub system built from Scribe [16]. The problem with their system is that it has some restrictions on the expression of subscriptions. Terpstra et al. [19] propose a content-based pub/sub system built on top of Chord. In order to have the system function correctly, it needs to maintain the invariants for filters in the face of node joins and departures, which is not an easy task. Triantafillou et al. [20] also built their content-based pub/sub system on top of Chord. However, the main drawback is that subscription installation and update may be expensive due to the large number of nodes and messages involved for attribute ranges in subscriptions. Reach [12] and HOMED [7] is a content-based pub/sub system built on top of a P2P overlay which maintains high-level semantic relationships. Both may have a load balancing issue since unevenly distributed subscriptions would cause unevenly distributed nodes in the overlay identifier space. In HOMED, it may be difficult to derive node IDs from their subscriptions while preserving high expressiveness of subscriptions and the change of a node's interests would cause the change of the overlay structure. Meghdoot [11] is based on CAN. 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's dimension is proportional to the number of event attributes.

Although Ferry is also built on top of a DHT, it differs from existing P2P-based solutions in that it exploits the embedded trees in a DHT to deliver events. Ferry's novel subscription installation and management algorithms allow the event delivery messages to be aggregated as much as possible along the dissemination paths, thereby avoiding redundant messages sent across the DHT identifier space. Moreover, Ferry provides a scalable and efficient platform to run any content-based pub/sub application with a unique scheme. It also supports the application with a very large number of event attributes.

3. System Design

In this section, we present the design of Ferry on top of Chord. However, the techniques discussed here are applicable or easily adaptable to other DHTs such as Pastry and Tapestry. It is worth pointing out that Ferry aims to serve as a platform to host many pub/sub services with unique schemes. For illustration purpose, we base our discussion on a pub/sub scheme $S = \{A_1, A_2, ..., A_n\}$, proposed by Fabret et al. [9]. In this scheme, a subscription is a conjunction of predicates over one or more attributes, and an example subscription is $s = \{(A_1 = v_1) \wedge (v_2 \leq A_3 \leq v_3)\}$. An example event is $e = \{A_1 = c_1, A_2 = c_2, ..., A_n = c_n\}$. In Ferry, each node serves as a rendezvous point (RP) for some subscriptions and events, and also as an intermediate node on the paths of event delivery. Given a 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).

3.1. Subscription Installation

A subscription s is represented by a pair (sid, p), where sid is a subscriber's node ID (*subscriber ID* for short), and p specifies a subscriber's interests in particular events by a conjunction of predicates which define the values or ranges

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

over one or more attributes in the scheme S. 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 a subscription. Due to space constraints, we omit the discussion of *RndRP* algorithm which aims to evenly distribute subscriptions over RP nodes. Please refer to our technical report [24] for more detail.

In this section, we present a more efficient algorithm, called *PredRP*, as outlined in Algorithm 1. 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*. As shown in Figure 1, r_1 is responsible for the subscriptions from $[r_1, r_2)$, and r_2 is responsible for the subscriptions from $[r_2, r_1)$. PredRP could achieve better performance than RndRP by avoiding sending the redundant messages across the Chord ring space. With RndRP, the event delivery messages from RP nodes r_1 and r_2 may need to traverse the whole Chord ring space since the subscriptions stored on each RP node may come from the subscribers distributed over 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 contiguous region of the Chord ring space (e.g., $[r_1, r_2)$ and $[r_2, r_1)$) respectively).

Algorithm 1 $install\_subscription$(Subscription s)

1: 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
2: $k = h(A_i)$
3: store s in a RP node which is an immediate successor node of k

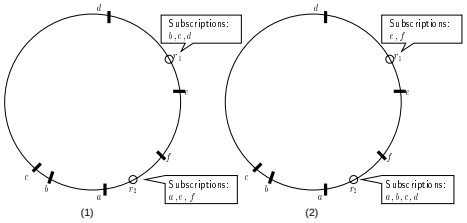

Figure 1. Illustration of RndRP and PredRP. r_1 and r_2 are two RP nodes. a, b, c, d, e, f are subscribers. (1) RndRP (2) PredRP.

When a subscriber leaves the system, he/she may unregister his/her subscriptions installed in the system, by simply requesting the corresponding RP nodes to remove his/her subscriptions. Otherwise, a subscription may have a TTL value. By associating a subscription with a TTL value, a subscriber does not need to unregister his/her subscriptions when leaving. The main drawback is that the subscriber needs to refresh his/her subscriptions periodically. However, the detailed discussion on unsubscribing is not focus of this paper.

3.2. Subscription Management

As discussed in [17], 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 rooted at the RP node r, say $EMDTree_r$ (an embedded tree for r) [2]. As will be discussed in Section 3.4, 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 subscribers? Recall that each Chord node's routing table consists of two parts: a *successor list* and a *finger table*. As outlined in Algorithm 2, r manages the subscriptions in a manner that a subscription s is stored according to the entry of a neighbor node (including both successor nodes and finger table nodes) *whose node ID is equal to or most immediately precedes s's sid* [3]. Note that this does not necessary 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, the discussion of how to associate subscriptions with routing table's entries is not focus of this paper.

Algorithm 2 $manage\_subscriptions$(Subscription s)

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

Figure 2 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 sid). 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. The subscriptions of subscribers z, w and v are stored corresponding to the entry of r's finger table node f_3. As will be shown later, this novel subscription management can allow a RP node to deliver events by aggregating messages along its DHT links (links to its successor nodes and finger table nodes), thereby reducing the number of messages across the system.

Now consider again Figure 1. PredRP may cause uneven subscription distribution across the RP nodes (r_1 stores less

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

[3] 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.

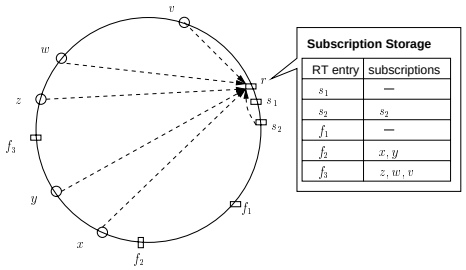

Figure 2. An illustration of a RP node r's subscription management. s_1 and s_2 are r's successors. f_1, f_2, and f_3 are members of r's finger table. s_2, x, y, z, w, and v are subscribers whose subscriptions are stored in r.

subscriptions than r_2). To deal with the load balancing issue, we propose a scheme, called *one-hop subscription push mode* (*one-hop push* for short). The basic idea is that a RP node (say r) may push the subscriptions corresponding to an entry of the finger table to the corresponding finger table node (say n). The RP node r then uses a *summary filter* [4] to represent the subscriptions pushed away. Upon an event e, r matches the event with the summary filter. If it is a match, the RP node r delivers e to the corresponding finger table node n (at this point, no subscriber ID list is carried in the event delivery message), which in turn serves as a RP node for those subscriptions pushed from r and starts delivering e to those matched subscribers.

One-hop push serves two main purposes. The first purpose is to move some of the loads (i.e., subscriptions and thereby subscription matching load) on a RP node 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. 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) and therefore the bandwidth cost. For more detail of one-hop push, please refer to our technical report [24].

3.3. Event Publication and Matching

When a node wishes to publish an event, the event is first directed to *all* of the RP nodes corresponding to the scheme S. These RP nodes are responsible for matching the event to the subscriptions and starting delivering the event to the matched subscribers. Given an event $e = \{A_1 = c_1, A_2 = c_2, ..., A_n = c_n\}$, Algorithm 3 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 publishing node has already cached the mapping of k_i to the IP address of the RP node. The direct point-to-point communication between the publishing 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 (e.g., in terms of bandwidth). We may need to use a more efficient mechanism to publish the event to the RP nodes instead, e.g, multicast techniques. More discussion of this is presented in [24].

Algorithm 3 $publish\_event$(Event e)

1: **for** each $A_i \in S$ **do**
2: $k_i = h(A_i)$
3: send e to a RP node which is an immediate successor node of k_i
4: **end for**

When an event e is published to a RP node r, r first needs to find the subscriptions matching the event, and then starts the process of delivering the event to the matched subscribers. Upon an event e, the RP node r needs to match e with the subscriptions it is storing. Algorithm 4 outlines the matching process which returns the list of matched subscribers with respect to the entry of r's k neighbor nodes.

Algorithm 4 $match\_subscriptions$(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}
1: **for** each neighbor node n_i **do**
2: **for** each subscriptions $s_j = (sid_j, p_j) \in store[i]$ **do**
3: **if** $is\_matched(e, p_j)$ **then**
4: $matched\_set[i].push\_back(sid_j)$
5: **end if**
6: **end for**
7: **end for**
8: return $matched\_set$

Note that Algorithm 4 is a linear subscription matching algorithm with respect to the number of subscriptions. To overcome this linear matching inefficiency, we could adopt sublinear matching algorithms based on building a subscription tree that collapses similar subscriptions [1]. However, how to optimize the matching algorithm is not focus of this paper. Algorithm 4 is primarily for illustration purpose.

[4] A summary filter in Ferry corresponds to an entry of the finger table and covers all currently hosted subscriptions for this entry of the finger table by exploiting covering relationships between subscriptions [22].

3.4. Event Delivery

Upon receiving an event, how does a RP node exploit the embedded tree in Chord to deliver events 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 5 and Algorithm 6 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 4) 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 matches 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.

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. Note that the event delivery algorithm in Ferry has several important features. First, no subscription matching operation is performed along the dissemination path except the RP node, due to the subscriber list contained in the message. Second, unlike MEDYM [3], it does not need to dynamically generate dissemination trees on-demand because it exploits the embedded trees which are inborn and dynamically maintained in Chord. Thirdly, the DHT link (or routing table) maintenance messages could be piggybacked onto the event delivery messages to reduce the maintenance cost which is inherent and nontrivial in DHTs.

Algorithm 5 $deliver\_event$(Event e, vector<ID> $matched\_set[1..k]$)

```
1: for i = 1 to k do
2:   if matched_set[i] is not empty then
3:     Message M ← e + match_set[i] {+ is a concatenation operator}
4:     send M to the neighbor node n_i, which then calls route_message(M) upon receiving M
5:   end if
6: end for
```

Algorithm 6 $route\_message$(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 sid_i ∈ list do
5:   if sid_i == this node's ID then
6:     deliver e to its local applications or users
7:   else
8:     find the neighbor node n_j whose node ID is equal to or most immediately precedes sid_i
9:     matched_set[j].push_back(sid_i)
10:  end if
11: end for
12: if matched_set is not empty then
13:   deliver_event(e, matched_set)
14: end if
```

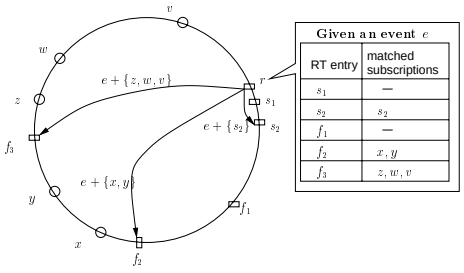

Figure 3. An illustration of the delivery of an event e from a RP node r. s_1 and s_2 are r's successors. f_1, f_2, and f_3 are members of r's finger table. s_2, x, y, z, w, and v are subscribers matching the event e.

4. Evaluation

In this section we evaluate Ferry using a scheme S for stock quotes application proposed by Gupta el al. [11]. We first describe the Ferry simulator, the scheme S for the stock quotes application, the datasets, and the metrics used for evaluation. Then, we present the experimental results.

4.1. Experimental Setup

We implemented Ferry on top of **p2psim** [5], a discrete-event packet level simulator. **p2psim** currently can simulate four P2P systems including Chord. 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 [8]. 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 [10]. The average round-trip time for the simulated network is 198 milliseconds.

The scheme S we used in our experiments was proposed in Meghdoot [11], and defined as $S = \{[Date : string, 2/Jan/98, 31/Dec/02], [Symbol : string, \text{“}aaa\text{”}, \text{“}zzzzz\text{”}], [Open : float, 0, 500], [Close : float, 0, 500], [High : float, 0, 500], [Low : float, 0, 500],$

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

$[Volume : integer, 0, 310000000]\}$. 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. Given the scheme S, an example subscription s is $(123456, \{(Symbol = \text{“}yhoo\text{”}) \wedge (High > 35.23)\})$, subscribed by a subscriber with $sid = 123456$.

We generated subscriptions by using five template subscriptions suggested in Meghdoot [11] with different probabilities. 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 which were modeled as exponentially distributed with an average inter-arrival time of 116 seconds.

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 headers, 33 bytes for the event, and 4 bytes for each subscriber ID carried in the message.

The results presented next do not include the 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 publishing 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, it can use Ferry's event delivery algorithm to publish the event to the RP nodes by envisioning the RP nodes as the publishing node's subscribers. In this case, the performance and cost of event publication is similar to event delivery.

4.2. Experimental Results

Due to pace constraints, some results are omitted here. Please refer to [24] for more detail. Table 1 shows the performance of Ferry with different configurations for $10,000$ subscriptions and $100,000$ events, where PNS represents Chord uses proximity neighbor selection (PNS). The average number of subscribers *per event* is 25, about 2.4% of nodes. Note that PredRP outperforms RndRP significantly, and PredRP+PNS performs the best. This is because (1) the event delivery messages in PredRP traverse shorter ranges of the 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 4 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 one-hop subscription push is very necessary for PredRP to achieve load balance. We also studied the impact of one-hop push on bandwidth cost for RndRP+PNS and PredRP+PNS. One-hop push (here it pushes the corresponding subscriptions of a RP node to its finger table nodes) could reduce 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 finger table 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.

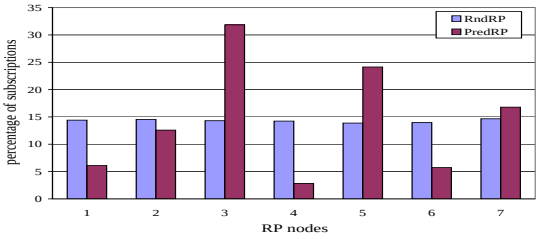

Figure 4. Subscription distribution in RP nodes for RndRP and PredRP.

To explore Ferry's performance (with configuration of PredRP+PNS) for various numbers 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. The number of subscribers varied from 2% to 80% of 1024 nodes per event. The results show that, as the number of subscribers increase, the hops and latency almost keep constant at 2.58 hops and 144ms, respectively. The bandwidth cost increases modestly, from 50.99 to 62.69 Bytes/node/event. However, the overhead drops significantly, from 1.36 to 0.02. The results show that Ferry could deliver events to a large number of nodes at very low overhead, involving only a small number of intermediate nodes by the synthesis of its message aggregation and PredRP algorithm.

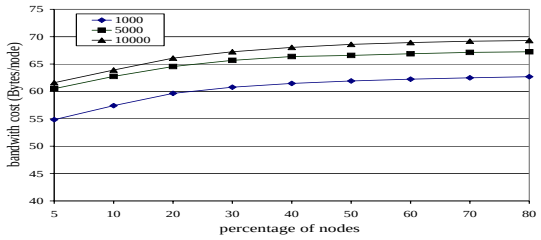

Figure 5. *bandwidth cost* for various network sizes with respect to different percentages of ndoes as subscribers.

We also investigated the performance of Ferry in various network sizes of 1000, 5000, and 10000 nodes. The simulated network of 5000 and 10000 were derived from the the 1024-DNS server measurements. For a give network size, we ran simulations for various percentages of nodes randomly chosen as subscribers (from 5% to 80%). We found the overhead is almost constant for various network sizes with respect to a given percentage of node as subscribers per event, from 0.86 to 0.02 when the number of subscribers varies from 5% to 80% of the system nodes. The number of hops taken by event delivery for network sizes of 1000, 5000, and 10000 are 2.58, 3.82 and 4.21, respectively. As the network size increases, the bandwidth cost incurred by event delivery increases modestly (as shown in Figure 5). This shows that Ferry can scale to a large number of nodes.

5. Conclusions and Future Work

Ferry is essentially a rendezvous network built on top of a DHT to support content-based pub/sub services/applications. Each application/service defines a unique scheme $S = \{A_1, A_2, ..., A_n\}$, and has at most n RP nodes (by hashing its attribute names). Subscriptions are routed to the RP nodes within $O(\log N)$ hops, while events are either directly sent to or routed to the RP nodes. Hence, events are guaranteed to meet all relevant subscriptions in the RP nodes. Ferry has several unique features: (1) Ferry exploits the embedded trees in a DHT to deliver events, thereby incurring little overhead; (2) the subscriptions are managed according to the entry of a RP node's routing table, thereby providing an efficient way to deliver events through aggregated messages; (3) event matching is performed only in the RP nodes (and their neighbor nodes if one-hop push is applied) by encapsulating the subscriber list in the event delivery messages; (4) its novel subscription installation algorithm, PredRP, can avoid sending redundant event delivery messages across the Chord ring space; (5) leveraging the fault-tolerance and self-organizing nature of DHTs, Ferry can reliably deliver events to subscribers and be fault-tolerant to node failures.

In Ferry, each node could be a RP node for some applications/services, and serve as the intermediate node to route events for other RP nodes. Hence, Ferry distributes the onus of event publication, event matching, and subscriptions across all nodes. Load balancing is based on the randomness guarantee of the consistent hashing function used in generating the RP nodes for different pub/sub applications/services (with different Ss). For a pub/sub scheme S with n attributes, the maximum number of RP nodes is n. All subscriptions of S will be stored in and all events will be routed to the n RP nodes. If the application/service corresponding to S is very popular, the subscriptions and events may overload the RP nodes. We therefore propose one-hop push to reduce the load of a RP node by moving part of its subscriptions to its finger table nodes. One-hop push may not work if a RP node's finger table nodes are all overloaded or not willing to take the load. Hence, we could adopt *attribute partitioning* [22] to address this issue. For example, consider 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. Note that with partitioning, we need to adapt the RndRP, PredRP, and event publication algorithms accordingly. Due to space constraints, we do not present the adapted algorithms here. However, the adaptation is very straightforward and simple.

In event publication, an event can be either directly sent or routed (by Chord routing protocol) to the RP nodes. If the number of the RP nodes (which is determined by the number of attributes of a scheme S and also subscription/event partitioning (if applied)) is small, the event publishing 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, e.g, 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 publishing node to a large number of RP nodes. Fortunately, Ferry's novel event delivery mechanism has already provided a solution for this problem, by envisioning the RP nodes as the subscribers of the event publishing node. Hence, Ferry can support a pub/sub scheme with a very large number of attributes.

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 P2P nodes have to be in Ferry (e.g., to provide incentive for nodes to cooperate in event delivery).

References

[1] 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)*, pages 53–61, Atlanta, GA, May 1999.

[2] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *Proceedings of the 19th IEEE ICDCS*, pages 262–272, 1999.

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

[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*, 19(3):332–383, 2001.

[5] 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.

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

[7] 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)*, pages 20–25, Edinburgh, Scotland, UK, May 2004.

[8] 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)*, pages 85–98, San Francisco, CA, Mar. 2004.

[9] 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*, volume 30, pages 115–126, Santa Barbara,CA, 2001.

[10] 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.

[11] 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.

[12] 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)*, pages 74–79, Edinburgh, Scotland, UK, May 2004.

[13] 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.

[14] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and Shenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM*, pages 161–172, San Diego, CA, Aug. 2001.

[15] 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)*, pages 329–350, Heidelberg, Germany, Nov. 2001.

[16] 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*, pages 30–43, 2001.

[17] 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*, pages 149–160, San Diego, CA, Aug. 2001.

[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] 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.

[20] 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)*, pages 104–109, Edinburgh, Scotland, UK, May 2004.

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

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

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

[24] Y. Zhu and Y. Hu. Ferry: An architecture for content-based publish/subscribe services on p2p networks. Technical report, Department of ECECS, University of Cincinnati, Oct. 2004.

[25] 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.