A DHT-based Infrastructure for Content-based Publish/Subscribe Services *

Xiaoyu Yang and Yiming Hu
Department of Electrical and Computer Engineering
University of Cincinnati, Cincinnati, OH 45221, USA
{yangxu,yhu}@ececs.uc.edu

Abstract

Publish/Subscribe model has become a prevalent paradigm for building distributed event delivering systems. Content-based publish/subscribe allows high expresses in subscriptions and thus is more appropriate for content dissemination. However, the scalability has remained a challenge in the design of distributed content-based publish/subscribe systems due to the expensive matching and delivering cost of content-based events. In this paper we propose an infrastructure built on top of distributed hash table for efficient content-based data distribution. Based on efficient subscription installation, event publication and event delivery mechanisms, the proposed infrastructure can simultaneously support any numbers of pub/sub schemas with different number of attributes. There are three key features in our design: (1) a locality-preserving hashing mechanism which partitions and maps the content space to nodes. Subscriptions and events are mapped to the corresponding node for efficiently matching; (2) an efficient event delivery algorithm which exploits the embedded trees in the underlying DHT to deliver events to the corresponding subscribers; (3) light-weighted load balancing mechanisms to adjust the load among peers and ensure that no peer is unduly loaded.

1. Introduction

The publish/subscribe (pub/sub) system has become a prevalent paradigm for delivering data/events from publishers (data/event producers) to subscribers (data/event consumers) across large-scale distributed networks in a decoupled fashion. In such a system, subscribers register their interests to the system using a set of subscriptions. The publishers can be completely unaware of the subscribers and simply submit information to the system using a set of publications. Once receiving a publication, the system matches it to the subscriptions and then delivers it to the interested subscribers. Traditionally, the designs of pub/sub systems can be typically classified into two classes: topic-based and content-based. Topic-based pub/sub systems need to predefine a set of topics. A subscriber registers a set of topics in which it is interested and then will be notified of all of the events associated with these topics. Relatively, content-based pub/sub systems allow users to flexibly specify complex interests with a set of predicates over the content of the publication, and thus are more appropriate for content dissemination.

Pub/sub systems can be either centralized or distributed. The centralized solution lacks scalability and suffers from single point of failure. Hence, the distributed solutions are more practical and preferred. The distributed hash tables [18, 16, 22, 15] paradigms are appropriate for building large-scale distributed applications due to their scalability, fault-tolerance and self-organization. As a result, many pub/sub systems [11, 6, 21, 20, 19, 13, 14, 1] have been built on top of DHTs. Peers in the system cooperate in storing subscriptions and matching & delivering events in a fully distributed manner. The main challenge in the implementation of DHT-based pub/sub systems involves: (1) the design of an effective subscription installation scheme that maps subscriptions onto peers in the system; (2) the design of an efficient event matching and delivery algorithm which distributedly filters and delivers events to the corresponding subscribers; (3) the design of load-balancing mechanism to ensure a uniform distribution of load among peers.

In this paper we propose a distributed architecture, built upon Chord [18], for content-based publish/subscribe services. Working as a scalable platform, the proposed infrastructure can simultaneously support many pub/sub schemas. For each pub/sub service, it employs a locality-preserving hashing mechanism to partition and map the content space to nodes. Subscriptions are mapped to one node or nodes that are close together in the overlay network, and events are delivered to the corresponding nodes for matching the subscriptions. Such a design can make the subscription installation and event publication efficient. Our architec-

*This research was supported in part by the National Science Foundation under grant CCF-0541103.
ture does not need to maintain dissemination trees to deliver events. Instead, it leverages the embedded trees in the underlying DHT for event delivery. Exploring DHT links can eliminate the overhead of maintaining additional in-network data structures for each pub/sub schema. Moreover, the maintenance of DHT links can be piggybacked onto the event delivery message, so as to reduce the maintenance cost. To deal with load balancing issues, we develop light-weight mechanisms to statically or dynamically adjust the load among peers, and ensure that no peer is unduly loaded.

The rest of this paper is structured as follows. Section 2 gives a survey of related work. Section 3 describes the key features of our design. Section 4 discusses load balancing issues. In section 5, experiments and results are presented and discussed. Finally, section 6 is the conclusion and future work.

2. Related work

Compared to the topic-based pub/sub systems (e.g. ISIS [4] and IBus [12]), content-based pub/sub systems are preferable since they give subscribers the flexibility to specify more complex interests with a set of predicates over the entire content of the publication. The events whose content satisfies all the specified predicates are delivered to the corresponding subscribers. As a result, subscriptions are more expressive but the system is harder to implement. Many content-based pub/sub systems are based on a spanning tree of all brokers (e.g. SIENA [5], Gryphon [2]). However, such a spanning tree is not feasible when the system involves a large number of brokers that join and leave the system dynamically. In addition, the overhead on brokers is so high that it may limit the system scalability and impose uneven load among nodes.

Structured P2P systems, such as Chord [18], Pastry [16], Tapestry [22], and CAN [15], use distributed hash table to construct the overlay network and provide more efficient lookups. Many attempts have been made to design a DHT-based pub/sub system [20, 21, 19, 13, 6, 14, 17, 23, 11, 1]. Scribe [17] and Bayeux [23] are topic-based pub/sub systems built on top of Pastry and Tapestry respectively. They can not directly support content-based pub/sub services. Tam et al. [19] built a content-based pub/sub system from Scribe. However, their system still suffers from some restrictions on the expression of subscriptions. Terpstra et al. [20] presented a content-based pub/sub system on top of Chord. In order to make the system function correctly, it needs to maintain the invariants for filters, which is inefficient in case of frequent node join and departure. Reach [13] and HOMED [6] are content-based pub/sub systems built on top of a P2P overlay, which maintain the high-level semantic relationships. They have a load balancing problem since unevenly distributed subscriptions will cause unevenly distributed nodes in the overlay identifiers space. Triantafillou et al. [21] built a content-based pub/sub system upon Chord. Content space for each attribute is mapped onto the ring. Subscriptions are stored onto the nodes whose identifiers lie in the corresponding range. The main drawback of their system is that subscription installation/reinforcement will involve a large number of nodes and messages. Aekaterinidis et al. proposed PastryStrings [1] on top of Pastry. Prefix string trees are maintained for content dissemination. The main limitation is that it could not efficiently cope with numerical range. [21, 1] match multi-attribute events through multiple processes on each attributes, and the final matched list is calculated from the results of the subprocesses. Such a method is very expensive in distributed environment. Meghdoot [11] is based on CAN. Considering the skewed distribution of real applications, it addresses the load balancing issue by zone splitting and replication. The main limitation is that the overlay’s dimension is twice of the number of event attributes, thus it can not effectively support multiple pub/sub services with different numbers of attributes. [21, 1, 11] focus on how to store the subscriptions and how to get the list of subscriptions matched to an event. However, how to deliver events to the matched subscribers is unexplored.

3. System design

In this section, we describe the design of our architecture on top of Chord. It should be noted that the techniques presented in this paper are applicable to other DHTs such as Pastry and Tapestry. We first give a brief description of the content-based pub/sub model, and then describe the locality-preserving hashing mechanism, followed by the discussions on the subscriptions installation and events delivery mechanisms.

3.1. Publish/subscription model

As proposed by Fabret et al. [9], a pub/sub schema can be described as: $S={A_1, A_2, \ldots, A_n}$, where each $A_i$ presents an attribute. Each attribute is defined by a name, a type and a domain. An event is a set of equalities on all attributes in schema $S$. A subscription is a conjunction of predicates on one or more attributes, where each predicate specifies a constant value or a range for an attribute. A subscription that needs to specify multiple predicates on the same attribute can be divided into multiple subscriptions. If a subscription does not specify any range over an attribute, the boundary of the domain of this attribute is considered as the interested range. Note that the prefix and suffix predicates on string type attributes can be converted to numerical ranges. An event $e$ matches a subscription $s$ if and only if
each predicate of \( s \) is satisfied by the value of corresponding attribute contained in event \( e \). Based on these assumptions and definitions, Content space of each pub/sub schema can be modeled as a multi-dimensional space, where each dimension represents an attribute. An event can be described as a point in the space, while a subscription is defined as a hypercuboid. An event matches a subscription if it is within the corresponding hypercuboid of the subscription.

### 3.2. Locality-preserving hashing

A key component in our design is a locality-preserving hashing (LPH for short) mechanism which partitions and maps a multi-dimensional content space to the 1-d key space. The mechanism is based on the technique of \( k \)-d tree [3]. The whole content space is partitioned into \( 2^m \) equal sized hypercuboids (\( m \) is the number of bits in the key identifiers of Chord), each of which is identified by a key (\( m \) bits integer).

Consider a \( d \)-dimensional content space \( \Omega \), where each dimension \( \Omega_i \) has a boundary described by an ordered pair \( \langle \Omega_i.L, \Omega_i.H \rangle \). The \( d \)-dimensional cuboids are obtained by dividing each dimension of \( \Omega \) alternately, for totally \( m \) times. The procedure satisfies the following two properties:

- After the \( i \)-th division, \( 1 \leq i \leq l \), \( \Omega \) is partitioned into \( \beta^i \) equal sized \( d \)-dimensional cuboids, where \( \beta \) is the base of the key/node identifiers \(^1\);
- The \( i \)-th division is performed on the \( j \)-th dimension, where \( j = i \mod d \).

The key is defined as follows: on the \( i \)-th division, if a hypercuboid picks up the \( p \)-th divided range from low to high on the splitting dimension, where \( 0 \leq p \leq \beta - 1 \), the \( i \)-th digit of the key is \( p \) (from the left, padded with zeros on the left if the length is less than \( m \)). Figure 1 illustrates the procedure of space subdivision and key generation on a 2-dimensional content space. Algorithm 1 describes the locality-preserving hash function. Given a point in the content space, it identifies the \( m \)-level hypercube which holds the content point and return the corresponding key as the hash value.

#### Algorithm 1 LPH (ContentPoint \( e \))

**Require:** \{\( \Omega \): domain of the content space; \( d \): the number of attributes in the content space; \( \beta \): base of the key identifier; \( m \): the number of digits in key identifier\}

1: \( T \leftarrow \Omega \)
2: \( \text{key} \leftarrow 0 \)
3: for \( i \leftarrow 1 \) to \( m \) do
4: \( j \leftarrow i \mod d \)
5: equally subdivide \( T_j \) into \( \beta \) ranges: \( r_0, r_1, \ldots, r_{\beta - 1} \), from low to high ordinarily
6: identifies the range \( r_p \) which holds \( e_j \)
7: \( T_j \leftarrow r_p \)
8: \( \text{key} \leftarrow \text{key} \times \beta + p \)
9: end for
10: return key

The Chord’s key-mapping mechanism is utilized to map the hypercuboids onto nodes in the system, i.e. each hypercuboid is mapped onto the node which is the successor (node whose identifier is equal to or immediately after the key along the ring) of its key. The above space partitioning and mapping mechanism can map nearby data points in the content space to one node or nodes close together in the overlay network, which makes the subscription installation and event publication efficient.

### 3.3. Subscription installation

In a pub/sub system, the subscription installation deals with the issue of efficiently storing subscriptions in the network. A user specifies his interest in particular events in the form of subscription which can be defined as a hypercuboid in the content space as discussed in section 3.1. Since the locality-preserving hashing mechanism partitions and maps the content space to nodes, a subscription \( s \) should be stored on the nodes whose content space overlaps the range of \( s \).

The major challenge in subscription installation is how to refine and route a subscription to the nodes whose content space overlaps the range of subscription. A naive approach is to subdivide a subscription into many sub-subscriptions, each of which is covered by only one of the \( 2^m \) hypercuboids, and route each sub-subscription to the corresponding node. This method is obviously inefficient and will cause high overhead especially when the range of a subscription is large. Accordingly, we use a different way to install subscriptions by progressively splitting and refining a subscription along the propagation path.

We define \( \text{prefix.key} \) and \( \text{prefix.length} \) to assist the subscription refining and routing. The \( \text{prefix} \) is the code (bit string) of the smallest hypercuboid that can completely hold the subscription region when the content space is recursively partitioned. The \( \text{prefix.key} \) is a \( m \)-bit identifier by padding zeros to the right of \( \text{prefix} \). The \( \text{prefix.length} \) is used to indicate the valid length of \( \text{prefix} \) in the \( \text{prefix.key} \). As illustrated in figure 2(a), the rectangle 011 is the smallest one that can completely hold the query region (the shaded

---

\(^1\)Generally, the base of key/node identifiers \( \beta = 2^b \), where \( b = 1, 2, \ldots \)
The subscription refining and routing algorithm is essentially a recursive process where subscriptions are progressively refined and delivered on the embedded trees formed by the DHT links. Thus, the overall number of messages needed to resolve and route a subscription can be significantly reduced.

Each subscription in the network is identified by an identifier subid, which is composed of the subscriber’s node ID nid and an internal ID iid which identifies different subscriptions of the same node. Given a subscription s, our infrastructure refines and delivers s to all the nodes whose content space overlaps with the range of s. We call these nodes the surrogate nodes of s. Subscription registration records the subid and range of subscription s to the repositories of s’s surrogate nodes.

### 3.4. Event publication and delivery

When an event is generated, the pub/sub system gets all matched subscriptions and delivers the event to the corresponding subscribers. As discussed in section 3.1, the event can be described by a point in the content space. Given an event e, the locality-preserving hash function is invoked to identify the m-level hypercuboid z which contains the event point e. We call content zone cz the rendezvous point of event e. The event e is then sent to the successor node of key(z), call node N, for matching the subscriptions stored on node N and delivering e to all of the matched subscribers. Algorithm 2 outlines the procedure of event publication.

**Algorithm 2 publish (Event e)**

1. key = LPH(e)
2. N = lookup(key) find the successor node of key
3. N.event.match.delivery(e)

Upon receiving an event e, node N matched e against all of the subscriptions stored on node N, a matched list of subids will be generated for event delivering, as shown in algorithm 3. The message for event delivery contains an event e and a corresponding subid_list.

**Algorithm 3 event.match.delivery (Event e)**

1. subid_list = match(e)
2. Message m = {e, subid_list}
3. event.delivery(m)

Upon receiving an event message m, a node, called Q, first extracts the subid_list from m, then divides subid_list into several subid_list by exploring node Q’s DHT links. All subids with targeting nodes sharing a common DHT link are put into the same list, according to Chord’s routing protocol. The message carrying each subid_list is then delivered through the corresponding DHT link. This mechanism can efficiently aggregate the event delivering messages and reduce the network bandwidth usage. Algorithm 4 outlines the procedure of event delivery which is essentially a distributed recursive process. Each node along the dissemination paths of the embedded tree processes and forwards...
event message until the event reaches all corresponding subscribers.

4. Load balancing

An important issue in the distributed system is load balancing. Each node in the system needs to store subscriptions and propagate events. Therefore, load on a node is due to both subscriptions and events. The locality-preserving hashing can not produce a uniform load distribution among nodes. The skewed distribution of real world dataset can cause a non-uniform distribution of load on the nodes. To this end, we propose load balancing mechanisms to statically or dynamically adjust load among nodes and ensure that no node is unduly loaded.

4.1. Space mapping rotation

Remember that our architecture can simultaneously support many pub/sub schemas. For each pub/sub schema, the locality-preserving hashing mechanism partitions the content space and maps content zones to nodes in the identifier space ranged $[0..\beta^m - 1]$. If each schema/subschema $S$ is given a random rotation offset $\phi$, content zone $cz$ of schema $S$ will be mapped to node $\text{successor}(key(cz)+\phi)$ (all arithmetic is modulo $\beta^m$). Thus content zones with identical key for different schemas/subschemas will be mapped to different nodes with a high probability. The mapping rotation mechanism can effectively distribute the large-size content zones for different schemas/subschemas onto different nodes. The randomness of $\phi$ for each schema/subschema can be achieved by hashing (with consistent hash function, e.g. SHA1) the name of the corresponding schema/subschema. The locality-preserving hash function can be easily modified to reflect the mapping rotation.

4.2. Content space transformation

The event load is not evenly distributed among the rendezvous point nodes when events are not uniformly distributed in the content space. Hence nodes in the events propagation path relative to hotspot rendezvous points may be overloaded due to event delivering.

For many real applications, the data distribution functions are monotonically increasing and can be modeled in advance based on the analysis of the characteristics and the historical data of the application. The original content space can be transformed to an intermediate space with data distribution approximately uniform according to the premodeled data distribution functions. Given a $d$-attribute content space $\Omega$, where each attribute $\Omega_i$ is bounded by $\langle \Omega_i, L, \Omega_i, H \rangle$ and conforms to a certain distribution with Cumulative Density Function (CDF) $F_i(x)$, the transform from $\Omega$ to $\Omega'$ satisfies the following two properties: (1) $\Omega'$ has same dimensionality and domain size as $\Omega$; (2) each data point $e$ in $\Omega$ is mapped to a point $e'$ in $\Omega'$ as follows:

$$
\begin{align*}
\begin{bmatrix}
\epsilon_0 \\
\epsilon_1 \\
\vdots \\
\epsilon_d \\
\end{bmatrix} & \Rightarrow \\
\begin{bmatrix}
F_i(\epsilon_0) \times (\Omega_i, H - \Omega_i, L) + \Omega_i, L \\
F_i(\epsilon_1) \times (\Omega_i, H - \Omega_i, L) + \Omega_i, L \\
\vdots \\
F_i(\epsilon_d) \times (\Omega_i, H - \Omega_i, L) + \Omega_i, L
\end{bmatrix}
\end{align*}
$$

The data space transformation leads to an approximately uniform distribution of events on content space $\Omega'$ and locality-preserving hashing on $\Omega'$ can produce a uniform distribution of event among rendezvous points.

4.3. Dynamic subscriptions migration

Subscriptions may not be evenly distributed among surrogate nodes. The overhead on heavily loaded nodes can be very high due to the storage cost of subscriptions, the CPU cycles for subscription management and event matching, and bandwidth cost for event delivery (the matched subid list can be very large). Here we propose a dynamic mechanism to migrate some load from heavily loaded nodes to lightly loaded ones.

At run time, each node periodically samples the load on its neighbors (and neighbors’ neighbors if the probing level $P_0$ is greater than 1). A node, called $N$, is said to be heavily loaded if its load oversteps the average load on the neighbors by a threshold factor $\delta_N$, that is, $L_N > L \times (1+\delta_N)$. The value of the threshold factor $\delta$ for each node is based on the node’s capacity. The average values of $\delta$ and $P_0$ control the tradeoff between the overhead and the quality of the load balancing. If node $N$ is overloaded, it will choose several lightweight loaded neighbors (or neighbors’ neighbors) and migrate some of its load to them. Without loss of generality, assume $N$ chooses $k$ nodes ($A_1, A_2, \ldots, A_k$) for load migration, and nodes $N, A_1, A_2, \ldots, A_k$ lie in the clockwise order on the Chord ring. Subscriptions with subscribers’ nodeIDs lying in the range $[ID(A_i)..ID(A_{i+1})]$ are migrated to node $A_i$, where $1 \leq i \leq k-1$. Subscriptions with subscribers’ nodeIDs lying in the range $[ID(A_k)..ID(N)]$ are moved to node $A_k$. Each node $A_i$ summarizes the accepted subscriptions and registers a surrogate subscription on node $N$.

The subscriptions migration mechanism can move some load from heavily loaded nodes to lightly loaded nodes. Thus the cost for storing, managing and matching subscriptions on the heavily loaded nodes can be significantly decreased. Moreover, subscriptions are migrated to nodes that are close (in the overlay) to the corresponding subscribers, therefore the average message size for event delivery can be reduced.
5. Performance evaluation

In this section, we evaluate performance of the proposed design through simulations. We start our discussion by describing the experimental setup and metrics used for evaluation. Afterwards, the experimental results are presented and discussed.

5.1. Experimental setup

We implement our pub/sub architecture on top of p2psim[7], a discrete event-driven, packet level simulator for many DHT protocols. We use Chord-PNS (Chord with proximity neighbor selection [8] allows each node to choose physical closest nodes from the valid candidates as routing entries, thus to reduce the lookup latency) protocol with its default parameters. The number of bits in the key/node identifiers in the simulator is 64. The network model used in the simulation is derived from the King dataset, which includes the pairwise latencies of 1740 DNS servers in the Internet measured by King method [10]. The average round-trip time of the simulated 1740-node network is 180 milliseconds.

We use synthetic datasets in our simulations. Events are generated based on Zipfian distribution, which is a common distribution of real world datasets. The cumulative distribution function for Zipfian distribution is $H_{k,s}/H_{N,s}$, where $H_{N,s}$ is the $N$th generalized harmonic number with skew factor $s$ and $1 \leq k \leq N$. Data points are modeled by scaling and shifting the domain of $k$. Subscriptions are generated from a template with the following properties: (1) the size of the range on each dimension is based on zipfian distribution; (2) the center of the range is based on the data distribution (same distribution as event points). The parameters and the default values of publish/subscribe model are list in table 1. The simulation starts by issuing 10 subscriptions on each node in the network. After system stabilization, we schedule 20,000 events generated on randomly chosen nodes. The interarrival time of these events is exponentially distributed with average value of 1000 milliseconds.

A set of cost metrics are used to evaluate the performance of the proposed architecture: (1) hops: the maximum path length required to deliver a subscription to all of the corresponding surrogate nodes and the maximum path length to deliver an event to all of the corresponding subscribers; (2) latency: the maximum time of delivering a subscription to its surrogate nodes and delivering an event to all of the corresponding subscribers; (3) bandwidth cost: total bandwidth consumption per subscription and the total bandwidth consumption for delivering an event to all of the corresponding subscribers. The size of each subscription message is modeled in byte as: $20 + 2 \times i_{\text{width}} + 9 + 1$, where 20 bytes are for package header, $i_{\text{width}}$ is the sum of the attributes' size, 8 bytes are for prefix_key, 1 byte is for prefix_length. The size of each event message is modeled in bytes as: 20 bytes for packet header, $i_{\text{width}}$ bytes for event, and 9 bytes for each $SubID$ (8 bytes for subscriber’s nodeID, and 1 byte for internalID) carried in the message.

5.2. Experimental result

We first evaluate the performance of subscription installation algorithm with respect to different range selectivity in the 1740-node network. We change the range selectivity on each dimension from 0.1% to 50%. As illustrated in figure 3, the overlay hops, latency and bandwidth cost increase when range selectivity increases. The overlay hops and latency increase slightly. Even the range selectivity increases to 50%, the overlay hops is still less than 10, and the maximum latency is less than 2100ms. The bandwidth cost has a reasonable increase when range selectivity becomes large because subscriptions with large range need to be delivered and stored to more surrogate nodes. (For 50% range selectivity, it only takes about 1.48Kbytes for delivering a subscription to large fraction of nodes in the 1740-node network.)

We evaluate the performance of event delivery algorithm in the 1740-node network. Figure 4(a) illustrates the distribution of events with respect to the percentage of matched subscriptions to all subscriptions in the system (the average value is 1.834%). Figure 4 (b) (c) and (d) depict the distribution of events with respect to hops, latency, and bandwidth cost respectively. The shape of curves in figure 4 (b) (c) and (d) close match the curve in figure 4(a). The average value of hops, latency, and bandwidth cost are 18, 1509ms, and 34.5Kbyte respectively (without load balance). Load balancing mechanism (we only evaluate Dynamic Subscription Migration mechanism in this paper) can effectively move some subscriptions from overloaded nodes the light-loaded ones, with slightly increasing the hops, latency and

<table>
<thead>
<tr>
<th>Dim</th>
<th>Size(byte)</th>
<th>Min</th>
<th>Max</th>
<th>Data skew factor</th>
<th>Data hotspot</th>
<th>Size skew factor</th>
<th>Size hotspot</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1000</td>
<td>0.2</td>
<td>9500</td>
<td>0.8</td>
<td>3%</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>-10000</td>
<td>10000</td>
<td>0.4</td>
<td>0</td>
<td>0.6</td>
<td>5%</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0</td>
<td>30000</td>
<td>0.4</td>
<td>100</td>
<td>0.5</td>
<td>10%</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>0</td>
<td>10000</td>
<td>0.1</td>
<td>5000</td>
<td>0.4</td>
<td>50%</td>
</tr>
</tbody>
</table>
The proposed load balancing mechanism can efficiently balance other load such as communication cost and CPU cycles etc. In this paper, we assume all nodes have same capacity (same threshold factors). We will evaluate the performance and cost for load balancing in heterogeneous environment with various parameters in the future.

### Table 2. Simulated Networks and Average RTTs

<table>
<thead>
<tr>
<th>Size ($10^3$)</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
<th>14</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ave RTT (ms)</td>
<td>174</td>
<td>176</td>
<td>182</td>
<td>182</td>
<td>180</td>
<td>180</td>
<td>178</td>
<td>178</td>
</tr>
</tbody>
</table>

We evaluate the performance in various sized networks (as listed in table 2) derived from king data. Since each node in the system generate 10 subscriptions, total number of subscription increases with the increasing of network size. It is hard to keep the average percentage of matched subscriptions unchanged for different network size. Figure 6(a) depicts the average percentage of matched subscriptions in each network. Although the average percentage of matched subscriptions decreases slightly with the increasing of network size, the average number of matched subscriptions per event increase largely when the number of nodes in the system increases. As shown in figure 6 (b), (c), and (d), the maximum hops, latency, and network bandwidth cost incurred by event delivery increase modestly as the number of nodes in the system increases from 2000 to 16000. This shows that proposed architecture can scale to large number.
of subscribers and large-size networks.

6. Conclusion and future work

In this paper we have proposed and evaluated the design of a scalable and decentralized infrastructure, which is built on top of distributed hash table for content-based publish/subscribe services. A locality-preserving hashing mechanism is developed to recursively subdivide the content space into layered content zones. Subscriptions and events are mapped to content zones for efficient matching. It does not need to maintain any dissemination trees for each pub/sub schema. Instead, it exploits the embedded trees in the underlying DHT for event delivery. Therefore the proposed architecture can simultaneously support many pub/sub schemas without the overhead of the maintenance of additional in-network data structures. We have also proposed light-weighted load-balancing mechanisms to adjust the load among nodes and ensure that no node in the system is unduly loaded.

This paper constitutes an initial step to build an efficient and scalable platform for supporting publish/subscribe services in peer-to-peer networks. There is much of future work to do. One is to enable the execution of real-world workloads and make the data distribution dynamically changed. Detailed evaluations will be performed on the subscription installation, event delivering and load balancing mechanisms, based on which more optimizations may be proposed. Currently, our design leverages the underlying DHT to deal with nodes join/departure/failure. However, the performance under high node churn rate has not been explored. This will be one of our future work.

References


