A Unified Construction Method of P2P Networks with Lower Bound to Support Complex Queries

Ye Yuan[1,2], Deke Guo[2], Guoren Wang[1], Yunhao Liu[2]
[1]Northeastern University, Shenyang 110004, China
[2]Hong Kong University of Science and Technology, Hong Kong SAR, China

ABSTRACT

The topological properties of overlay networks are critical for the performance of peer-to-peer (P2P) systems. The size of routing table and the diameter are among the most important parameters which measure the autonomy, efficiency, robustness and load balancing of P2Ps, and Moore bound is optimal tradeoff between diameter and degree for any graph. In order to improve the four features for structured P2P networks, researchers have proposed many optimal designs with desirable degree and diameter. Most systems are based on interconnect networks which approximately achieve the Moore bound. Moore bound, however, cannot give a well description for P2P systems due to their dynamic features. In this paper, therefore, we first calculate a new lower bound of the network diameter and average query distance in a high dynamic environment. In addition, the existing systems require the number of peers to be some given values determined by the degree and the diameter. Hence, existing designs cannot address the issue due to the fact that a P2P system is typically dynamic with peers frequently coming and leaving. To solve the issue, we then propose a universal method based on the multi-way trie tree to address the inefficiency caused by peers frequent coming and leaving. Finally, our design - Phoenix, based on the multi-way trie tree and deBruijn structure, can well support exact query as well as complex queries, such as range queries, multi-attributes queries and multi-dimensional data queries. Also, the method proposed for Phoenix can also be applied to other interconnection networks after minimal modifications, for example, hypercube, butterfly, CCC and kautz.

1. INTRODUCTION

The growing popularity of P2P networks makes them very likely substrate for future internet scale information systems. Following Napster and Gnutella-like networks, several structured P2P networks based on distributed hash tables (DHTs) [1, 2, 3, 4, 5] have emerged. After that, more developments [6, 7, 8, 9, 10, 11] are introduced which provide great scalability, accuracy and efficiency.

Two important properties often indicate the quality and efficiency of structured P2P networks: (1) peer degree, the size of routing table to be maintained on each peer, and (2) network diameter, the number of hops a lookup needs to travel in the worst case. In traditional structured P2P designs, the peer degree and network diameter increase logarithmically with respect to the size n of the network, such as Chord [3], Pastry [1] and Tapestry [2] based on the hypercube topology, and SkipNet based on skip list structure [12]. Such schemes publish and lookup resources within $\log n$ hops, while they often introduce huge maintenance overhead and suffer from poor scalability.

To address these problems, researchers propose several architectures based on static interconnection networks. For examples, the Viceroy [9] and Ulysses [11] are based on the butterfly topology [9], Cycloid [13] is based on the CCC topology [14], CAN [1] is based on the d-dimensional torus topology, Koorde [7], Distance Halving [10], D2B [6] and ODRI [8] are based on the de Bruijn topology, and FissionE [15] is based on the Kautz topology. In above designs, the network diameter increases logarithmically with respect to the size of a P2P system.

Moore bound is optimal tradeoff between diameter and degree for any above static graph. Moore bound, however, cannot give a well description for P2P systems due to their dynamic features. In this paper, therefore, we need reconsider the lower bound. We calculate a new lower bound of the network diameter and average query distance in a high dynamic environment.

One critical requirement of most existing P2P systems is that the number of peers must be some given values determined by the peer degree and the network diameter. Hence, the corresponding approaches are often impractical when peers frequently join, leave, and fail. Several systems designed recently support arbitrary number of nodes, however, the maintenance overhead is usually huge. In this work, we aim at giving a universal method for any P2P systems with arbitrary number of peers, so that the method is easy to implement in practice. It is well known that the traditional DHTs cannot support complex query schemes well. Our new system, Phoenix, possesses all the good properties of static network, as well as support complex queries and the exact query in the dynamic environment.

The main contributions of this paper are as follows: (1) We calculate a probabilistic lower bound of the network diameter and average query distance in a high dynamic environment. (2) We propose a multi-way trie tree structure whose nodes located at same level form a ring structure. Any static interconnection network can be embeded in the ring structure. Based on this universal method, an existing static network can implemented as a overlay topology of a dynamic P2P system. (3) Based on the multi-way trie tree and deBruijn graph, we design Phoenix: an effective and robust P2P system which retains desirable properties of static deBruijn digraph, such as optimal diameter and constant out-degree. We also design a resource placement strategy, an effective and robust routing algorithm, and range and multi-attribute queries algorithms. This design is able to deal with the dynamic operations of peers, such as join, depart, fail, and topology changes. (4) We evaluate the properties of Phoenix, the robustness of routing scheme, the delay and message costs of basic operations and load balance by formal analysis and simulations, and compare it with previous P2P designs.

2. RELATED WORK

Some existing DHTs usually predefine some parameters as the interconnection networks do, for example, the number of nodes. Those systems are static, and not suitable for dynamic network. In addition, the identifiers of peers must be the same length. The requirement, however, cannot guarantee the connectivity of a DHT in a dynamic environment. To address this problem, some systems relax the interconnection network protocol at the cost of other negative impacts, such as uneven distribution of node degree and less efficiency of routing protocol. For example, Tapestry modifies the hypercube connection protocol, and allows each node keep almost $n/2$ nodes as neighbors. To overcome those disadvantages of previous relaxation methods, we propose a universal method for any static graph. This method can result in a distributed system which can maintain the shape of related static graph well.

Range query is one of the most important query over single attribute, multi-attributes, multi-dimension data. The most generic proposals so far are the trie-based prefix hash tree (PHT) [27] and distributed segment tree (DST) [28]. However, such approaches have common disadvantage: they can keep the locality of data in each range, but cannot preserve the locality of consecutive ranges. And hence, a query coving a continuous ranges cannot been resolved in an efficient way because multiple overlay network queries are required to locate all the content. Andrzejak and Xu extends CAN to support single-attribute range query. They use the Hilbert Curve to represent an whole interval of attribute values, and then map the resulted curve over a Content Addressable Network (CAN). Tanin et al. use a MX-CIF quadtrees [29] to index complete data such as spatial data. MAAN extends Chord to support multi-attribute range queries [30] by replacing the uniform randomized hash function with a locality preserving hash function.

BATON [21] is based on a binary balanced tree which can support range queries. VBI [23] extends BATON to support multidimensional data objects. It follows the design principle of BATON and suffers from the update and reconstruction of index structures, such as R-Tree, M-Tree, and SS-Tree. The binary tree, however, provide large search delay. Mercury [22] provides probabilistic guarantees on search and load-balancing, even when the P2P system is fully consistent. Mercury also stores data continuously in order to support signal and multi-attribute range queries. BATON* [24] and P-Ring [25] provide search delay $\log n$, but the performance is not optimal. For example, the search delay of BATON* is $log_d n$ while its degree is $dlog_d n$. The search delay of Phoenix we proposed can achieve optimal $\log n / \log\log n$, when the peer degree is $\log n$. On the other hand, those systems cannot adapt to the highly dynamic environments since they must execute an expensive load balancing algorithm frequently to balance storage load. Phoenix can keep the efficiency of a query on a uneven distributed data set, and its load balancing algorithm only affect a small portion of peers.

The rest of the paper is organized as follows. First of all, the search delay is analyzed for the dynamic network, then two lower bounds are proved for the network diameter and average query distance, respectively in Section 3. Then a universal method is proposed to solve the issue of the static network in Section 4. A new P2P system - Phoenix based on the universal method is proposed in Section 5. Finally, the performance of Phoenix are evaluated in Section 6 and this paper is concluded in Section 7.

3. A LOWER BOUND OF QUERY DELAY IN DYNAMIC P2P NETWORK

Most systems introduced are all static, and they cannot adapt to the dynamic network. The negative effects due to possible dynamic behaviors should been considered for topology properties in all static networks. In this section, we will discuss the impact on the diameter and average query distance.

The problem of designing an optimal-diameter graph of fixed degree has been extensively studied in the past. In one formulation of this problem, assume a graph of fixed degree and diameter. What is the maximum number of nodes that can be packed into any such graph? A well-known result is the Moore bound:

$$n \leq 1 + k + k^2 + ... + k^d = \frac{k^{d+1} - 1}{k - 1} \qquad (1)$$

In the context of P2P DHT, we are concerned with a different formulation of the problem: given nodes and fixed degree, what is the minimum diameter in any graph built on top of these N nodes? The answer follows from (1):

$$d \geqslant \lceil \log_k (n(k-1)+1) \rceil - 1 \qquad (2)$$

When k is $\log_2 n$, the lower bound is $O(\frac{\log_2 n}{\log_2 \log_2 n})$. When k is a constant, the lower bound is $O(\log_k n)$. It is well known that Moore bound is a lower bound of diameter for any non-trivial graph. It is satisfied in the dynamic environments, but its value is too small for dynamic networks. Moore bound for static networks cannot give a well description for dynamic networks. We, therefore, need reconsider the lower bound.

In [16], authors analyze the load balancing problem by a binary trie tree model. A binary tree is two way tree with different length strings. Our idea comes from this model, since a routing tree can be used to emulate dynamic network. Without loss generality, we set up a d way tree in our model.

For a static network with maximum out-degree d, any node cannot reach more $d + d^2 + ... + d^h$ nodes in at most h hops. Hence, in order to reach all $n-1$ nodes in at most h hops, the value of h should satisfy $d + d^2 + ... + d^h \geq n - 1$. But in a dynamic environment, the number of nodes reached from a beginning node in i hops may not be d^i nodes, for $1 \leq i \leq h$. In other words, the routing tree may not be completed in each level.

To emulate a dynamic network, let us consider a case that nodes join in and leave randomly from a trie tree. The joining process of a node is equivalent to that a ball with a prefix x being dropped into the root of a virtual tree and then reach a most matched node with x of d children as the next step. The leaf at which the ball ends up is the node's parent, and shares the longest common prefix with x. The joining process is similar as the most existing DHTs hold. Peer departure is a opposite process of peer joining. In a static network, each node joins the network in a given order. After the $k-1$th level is full (there are d^k nodes in the level), a new node joins into the kth level. However, the nodes close to the root nodes always suffer from high traffic overhead due to peer joining operation, especially when large number of nodes want to join the network.

Now we can estimate a lower bound of the diameter for a dynamic network. Here, the diameter is defined as the largest depth of the tree.

Theorem 1. *With probability* $1-O(1)$*, the diameter lower bound* d_{max} *of dynamic network concentrates on three values* d_l*,* d_l+1*, and* d_l+2*, where*

$$d_l = \lfloor \log_d n + \sqrt{d \log_d n} - 1.5 \rfloor \tag{3}$$

Proof: Since d_{max} is the maximum height of the tree, to compute d_{max}, we need to consider the leave process of a node. This process is constructed by dropping balls and getting rid of leaves. The process is very similar with the well know structure called the Patricia trie [17]. This tree is a collapsed version of the regular trie in which all intermediate nodes with a single child are removed. Recalling that with probability $1-o(1)$, the height of a random Patricia trie is concentrated on three values d_l, d_l+1, and d_l+2 , where d_l is given by (3), we immediately get the statement of the theorem [17]. ■

The diameter of a graph is simply the largest distance between any pair of nodes and only provides an upper bound on the delay (number of hops) experienced by users. A much more crucial metric is the average distance between any pair of nodes. We will focus on a lower bound of this metric in a dynamic network.

Let h denote the minimum depth of the d way tree. The tree is divided by two parts. First part consists of all nodes which locate between the root and $h-1$ level. Second part includes all nodes which locate between level h to $d_{max}-1$. Naturally, the first part is a full d way tree with depth h. Let $d(root, x)$ (denoted by d_x) denote the distance from the root to any node x in the tree. We will discuss the distribution of d_x for the two parts separately.

To compute the distribution the first part, we define a sequence of indicator random variables A_i, $i \geq 0$, where $A_i = 1$ if level i of the tree is full after users joined the system and $A_i = 0$ otherwise. We say that a level is full if all nodes of that level are present and nonleaf. Notice that level i can be full only if $i < d$ and that $A_i = 1$ implies that $A_k = 1$, for all $k < i$. It immediately follows d_x is at least $k+1$ if and only if all levels from 0 to k are full:

$$P(d_x \geq k+1) = P(\bigcap [A_i = 1]) \tag{4}$$

We then have the deduction of formula (4):

$$P(d_x \geq k+1) = P(d_x \geq k) P(A_k | A_{k-1}) \tag{5}$$

where $P(d_x \geq 0) = 1$ and $P(A_k|A_{k-1})$ is the conditional probability of level k being full given that all previous levels $0, ..., k-1$ are full:

$$P(A_k|A_{k-1}) = P(A_k = 1 | h \geq d_x) \tag{6}$$

For the first part, we can draw out the following theorem.

Theorem 2. *With probability* $1-O(1)$*, the distribution of* d_x *is* $\exp\{-d^k \exp\{\frac{n(d-1)+1}{d^{k+1}(d-1)} - \frac{1}{d-1}\}\}$.

Proof: First notice that the first part is a d way tree with h level. The d way tree built using peers contains $\frac{n(d-1)+1}{d}$ leaves and nonleaf $\frac{n(d-1)+1-d}{d(d-1)}$ nodes. Next, examine level k of the tree and observe that all d^k possible nodes at this level must be non-leaf for level k to be fully joined. Assuming that all previous levels are full, exactly $\frac{d^{k-1}}{d-1}$ non-leaf nodes contributed to filling up levels $0, ..., k-1$ and the remaining $\frac{n(d-1)+1-d}{d(d-1)} - \frac{d^{k-1}}{d-1} = \frac{n(d+1)+1-d^{k+1}}{d(d-1)}$ non-leaf nodes had a chance to be joined. After the first levels $k-1$ have been filled up, each node at level k is hit by an incoming ball with an equal probability d^{-k}. Thus, our problem reduces to finding the probability that $u = \frac{n(d+1)+1-d^{k+1}}{d(d-1)}$ uniformly and randomly placed balls into $m = d^k$ bins manage to occupy each and every bin with at least one ball. There are many ways to solve this problem, one of which involves the application of well-known results from the coupon collector's problem [18]. We use this approach below. Define $Z(u)$ to be the random number of non-empty bins after balls u are thrown into m bins. Thus, we can write $P(A_k|A_{k-1}) = P(Z(u) = m)$. Recall that in the coupon collector's problem, u coupons are drawn uniformly randomly from a total of m different coupons. Then, the probability $Z(u) = m$ to obtain m distinct coupons at the end of the experiment is given by [18].

$$P(Z(u) = m) = \sum (-1_j)(m/j)(1 - \frac{j}{m})^u \tag{7}$$

For large u, the term $(1-j/m)^u$ can be approximated by $e^{-uj/m}$, yielding

$$P(Z(u) = m) \approx \sum (-1_j)(m/j) e^{-uj/m} = (1 - e^{-u/m}) \tag{8}$$

Since we are only interested in asymptotically large $m = O(log_d n)$, (8) allows a further approximation

$$\begin{aligned} P(Z(u) = m) &\approx e^{-me^{-u/m}} \\ &= \exp\{-d^k \exp\{\tfrac{n(d-1)+1}{d^{k+1}(d-1)} - \tfrac{1}{d-1}\}\} \end{aligned} \tag{9}$$

From (7)-(9), we get the distribution of d_x:

$$\begin{aligned} P(d_x \geq k+1) &= P(d_x \geq k) P(A_k | A_{k-1}) \\ &\approx \exp\{-d^k \exp\{\tfrac{n(d-1)+1}{d^{k+1}(d-1)} - \tfrac{1}{d-1}\}\} \end{aligned} \tag{10}$$

■

From (10), we know that

$$\begin{aligned} h &= max(d_x) \\ &= \log_d n - \log_d((1+\varepsilon)\log n - O(\log\log n)) \end{aligned} \tag{11}$$

with probability at least $1 - n^{-\varepsilon}, \varepsilon \leq 1$.

For the second part of the tree, we have the following theorem from [17].

Theorem 3. *With probability* $1-O(1)$*, the distribution of* d_x *of PATRICIA tries is:*

$$d_x \sim \sqrt{1 + d\xi\Phi'(\xi) + \xi^d \Phi''(\xi)} e^{-n\Phi(\xi)} \tag{12}$$

where $\xi = nd^{-k}$*,* $0 < \xi < 1$*, and*
$\Phi(\xi) \sim \frac{1}{d}\rho_0 e^{\varphi(log_d \xi)} \xi^{3/2} \exp(-\frac{\log^d \xi}{d \log d})$.

From the distribution of d_x for two parts of the tree, we can get the lower bound of average distance for a dynamic network. We have the following theorem.

Theorem 4. *The lower bound of average distance for a dynamic network is* $log_d n + \sqrt{dlog_d n}/log_d \log n$ *with probability at least* $1 - n^{-\varepsilon}, \varepsilon \leq 1$.

Proof: From (9)-(12), we get the average distance: $d_{avg} = \sum_{k=0}^{h-1}$ $(\exp\{-d^k \exp\{\frac{n(d-1)+1}{d^{k+1}(d-1)} - \frac{1}{d-1}\}\})+ \sum_{k=h}^{d_{max}}$ $(\sqrt{1 + d\xi\Phi'(\xi) + \xi^d\Phi''(\xi)}e^{-n\Phi(\xi)}) \approx log_d n + \frac{\sqrt{dlog_d n}}{log_d \log n}$ ■

So far, we derive the lower bounds of diameter and average distance for a dynamic network. The search delay of any existing P2P system cannot be smaller than the lower bound *w.h.p.*

Since a dynamic network may incur randomness, the diameter and average distance may become wobbly. We should reduce the randomness so as to make the delay close to a static value. In next section, we will propose a universal method to avoid the randomization problem.

4. A UNIFIED CONSTRUCTION METHOD FOR DYNAMIC P2P NETWORKS

In this section, we propose a unified construction method for any static network. Hence, a static network can adapt to dynamic environments. We select DeBruijn as an example because it is one of the best static structure for P2P networks [8]. We introduce a new P2P system, Phoenix, based on deBruijn. This method can be applied to other static networks after minor modifications. Phoenix is a completely distributed system with arbitrary number of peers, and has a good scalability in dynamic environments.

4.1 Dynamic Multi-way Trie Tree Structure

DeBruijn graph is an optimal topology only if all nodes exist and are stable. This requirement, however, is impractical under the dynamic scenario. To address this issue, we propose a dynamic multi-way trie tree structure to achieve the desired topology.

Definition 1. *A d way trie tree with depth k is a rooted tree. Each node has at most d child nodes. Each edge at same level is assigned a unique label. Each node is given a unique label. The label of a node is the concatenation of the labels along the edges on its root path. The label of each edge is assigned based on the following rules.*

1. The edge from the root node to its ith child is labeled as $x_1^i = i$ for $0 \leq i \leq d-1$. The ith child of root node is labeled as $x_1^i = i$, and is arranged from left to right. The root node dose not contain any string.
2. The edge from a node x_1 to its ith child is labeled as $x_2^i = i$ for $0 \leq i \leq d-1$. The ith child is labeled as $x_1x_2^i$, and is arranged from left to right.
3. The edge from a node $x_1x_2...x_{k-1}$ to its child is labeled as x_k^i. The child of x_k^i is labeled as $x_1x_2...x_{k-1}x_k$, and is arranged from left to right.
4. There is a bidirectional edge between any node and its parent, and all the nodes at the same level form a ring.

From the definition, we know each node is assigned a unique identifer. The length of an identifier in level k is longer one than that in level $k-1$. The MD5 algorithm will be used to generate a unique identifier for each node when the node joins Phoenix. After the node fixes its level, the node can truncate the string to ensure the rule of the tree. The level of the root node is 0, and its child nodes locate at level 1, and so on. In general, the length of the label of each node shows its level in the tree. Fig. 5 shows an example of a two way trie tree.

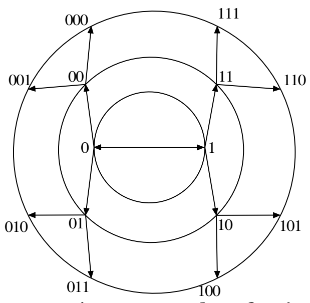

Figure 1: An example of trie tree

The tree might be imbalanced due to the use of MD5 algorithm. This issue, however, does not affect our system as discussed in the following section. In fact, it is impractical to keep a balanced tree in a dynamic scenario. Definition 2 gives a definition of the balanced tire tree which also be called as dynamic d way trie tree.

Definition 2. *A d way trie tree with depth k is balanced if all leaf nodes are at the level k. A balanced trie tree is a complete trie tree only if the parent node of any leaf node has full child, otherwise it is an incomplete trie tree or dynamic trie tree.*

The tree shown in Fig. 1 is a complete trie tree. If the leaf node 011 fails, the tree becomes an incomplete trie tree. If the node 010 and 011 both fails, the tree becomes a unbalance tree.

For any level, the left-to-right traversal of nodes at the level can form a total ordering, denoted as the trie ordering, in a straight forward manner. Let's sort all the child nodes of root node in ascend order within level 1, and then rank all the child nodes of each node at level 1 respectively. Note that the nodes within level 2 inherit the order of nodes within level 1. By sorting nodes from level 1 to k recursively, we can find a trie ordering of nodes at each level. From the trie tree structure, we know each level forms a ring and we call the ring trie ring. We arrange the trie ordering in each ring along the anti-clockwise direction. In the ring, any node connect its predecessor and successor.

Definition 3. *For any node x, its predecessor is the first existing node clockwise from it in a trie ring of existing nodes at same level, and its successor is the first existing node anti-clockwise from it in the same trie ring. The concept of left adjacent node is similar to the predecessor, but the trie ring is consisted of all possible nodes not just existing nodes. So do the right adjacent node and successor node.*

In a complete trie tree, the left adjacent node and predecessor are the same nodes. So do the successor. But in an unbalanced tree, they may not be the same nodes. Therefore we should establish the successor and predecessor of a node for both the balanced and unbalanced tree. In the figure, the node 010 is the predecessor of the node 011 and its successor is the node 100. In the unbalanced tree, (the node 010 and 100 are not alive) the predecessor of the node 100 is the node 001 and 001's successor is 100.

Algorithm 1 can establish the successor of a node for both a balanced and an unbalanced tree. For the algorithm, $x = x_1x_2...x_k$ is a peer of the trie tree. x express the same meaning in other algorithms.

For a complete tree, it's easy for a node to establish its successor only according to its identifier. Surely, a node of a balanced tree can also use Algorithm 1, and the leaf node

Algorithm 1 Successor(x)

```
1: $i \leftarrow k$
2: do{
3: $x_1x_2...x_i$ forwards the request to $x_1x_2...x_{i-1}$ using its up link.
4: $i \leftarrow i-1$
5: }while(the up link to $x_1x_2...x_i$ is its last down link and
   $x_1x_2...x_i$ has not fixed its successor)
6: $j \leftarrow i$
7: if ($x_1x_2...x_j$ has fixed its successor) then
8:    do{
9:    $x_1x_2...x_j$ forwards the message including the identifier of
      its successor $x_1x_2...x'_j$ to $x_1x_2...x_{j-1}$.
10:    if ($x_1x_2...x'_j$ has a child) then
11:       $x_1x_2...x_{j-1}$ initiates a request to $x_1x_2...x'_j$ to fetch its
          first child as its successor.
12:    else
13:       return false.
14:    end if
15:    $j \leftarrow j-1$
16:    }while($j > k$)
17: else
18:    $x_1x_2...x_j$ returns its first child as the successor of
       $x_1x_2...x_{j-1}$.
19:    $m \leftarrow j-1$
20:    do{
21:    $x_1x_2...x_m$ forwards the message including the identifier of
       its successor $x_1x_2...x'_m$ to $x_1x_2...x_{m-1}$.
22:    if ($x_1x_2...x'_m$ has a child) then
23:       $x_1x_2...x_{m-1}$ initiates a request to $x_1x_2...x'_m$ to fetch its
          first child as the its successor.
24:    else
25:       return false.
26:    end if
27:    $m \leftarrow m-1$
28:    }while($m > k$)
29: end if
```

at most forwards the request to its grandparent. But in dynamic scenario, the tree is usually unbalanced. At this time, we should use Algorithm 1.

For a unbalanced tree, there are two steps to perform Algorithm 1. In the first step, x forwards the request along the up link. When an ancestor y has established its accessor, the request stops. When an ancestor z has at least two children and requesting link is not the last children (As described above, we have sorted all the children of a node according to their keys.), its immediate right sibling is the successor of the requesting child. At this time, the request stops. In the second step, the ancestor returns the information got in the first step back to its requesting child, and its child can establish its successor according to the information. At last, all the ancestors from x to y or z establish their successors.

Theorem 5. *With probability* $1-O(1)$*, the cost of establishing a successor for a node in a unbalanced trie tree is at most* $2\lfloor\sqrt{d\log_d n} + 0.5 + \log_d((1+\varepsilon)\log n - O(\log\log n))\rfloor$.

The proof can be found in our technique report [26]. In this paper, we just present the most important results and omit technical details.

In fact, Algorithm 1 also establishes the predecessors of many nodes, since a node is the predecessor of its successor. If many nodes have established their successors, the cost of establishing a successor and a predecessor for a new node will be very low. Meanwhile, it is easy to establish the predecessor of a node, and vice versa. For almost the same principle, we can establish the predecessor of a node. The details are omitted due to the limitation of the space.

5. PHOENIX

In this section, we will propose desirable structuring strategies to organize peers in an efficient overlay network that can guarantee logarithmic network diameter and constant degree of each peer. All the structuring strategies are based on the concept of d way dynamic trie tree with depth k mentioned above. First, each peer is mapped to exactly one node in the trie tree, and uses the identifier of its related node and IP address as its logical and physical identifiers, respectively. Second, deBruijn graph is embed in each level. Each inner peer of the tree maintains $2d+3$ neighbor peers and each leaf peer maintains $d+3$ neighbor peers. Third, any resource achieves an identifier from a similar identifier space as that of peers, and be distributed at a given leaf peer based on the longest prefix matching rule. The identifier of a resource is directly used from the 128 bit hash value space. Finally, the topology rule, naming rules of peer and resource, and resource distribution rule are used to instruct how to route a message between any pair of source and destination peers.

5.1 Topology construction and resource placement rule

P2P overlay network must support arbitrary number of peers in order to deal with the uncontrolled dynamic operations of peers, such as joining, departure and failure. Unfortunately, an overlay network based on a static network works well only if number of peers equals to a series of discrete numbers. For example, for a d-ary deBruijn graph, the number of peers should be $d, d^2, d^3, ..., d^k$. To address this issue, we propose a general overlay network, Phoenix, based on the d way dynamic trie tree. Any static network introduced in Section **??** can be embedded in the trie tree so that the efficient routing algorithm of a static network can be proposed for dynamic network. In this paper, we choose d-ary deBruijn graph as an example.

The d-ary deBruijn graph is embedded in the d way trie tree as follows:

For a peer $x = x_1x_2...x_k$, $\sigma^i(x)$ denotes an operation such that $\sigma^i(x) = x_2...x_kx^i_{k+1}$ for $0 \le x_{k+1} \le d-1, 1 \le i \le d$. $\sigma^i(x)$ is called the out-neighbor of peer x, and x is called the in-neighbor of peer $\sigma^i(x)$.

(1) If a peer $\sigma^i(x)$ has appeared in the overlay network, it is the ith neighbor of peer x;

(2) Otherwise, if $\sigma^i(x)$ and its predecessor y have a common prefix with length $k-1$, peer y is the ith neighbor of peer x and y stores another identifier $\sigma^i(x)$.

(3) Otherwise, if $\sigma^i(x)$ and its successor z have a common prefix with length $k-1$, peer z is the ith neighbor of peer x and y stores another identifier $\sigma^i(x)$.

(4) Otherwise, the youngest living ancestor of $\sigma^i(x)$ is the ith neighbor of peer x and the ancestor stores another identifier $\sigma^i(x)$.

For example, in Fig. 1, peer 010 connects to peer 100 and 101. If peer 100 fails, peer 101 replaces 100. As a result, peer 010 connects to 101 again and peer 101 stores the identifier 100. If peer 100 and 101 both fail, peer 010 connects to their parent 10 and peer 10 stores the identifier 100 and 101.

From the construction rule, we can draw the following conclusion.

Theorem 6. *For an inner peer x and leaf peer y of Phoenix, the former rules can guarantee the following conclusion:*

(1) For a balanced tree, x has $2d+3$ neighbors and y has $d+3$ neighbors.

(2) For an unbalanced tree, x has at least $2d+1$ neighbors and y has at least $d+1$ neighbors

The proof can be found in our technique report [26].

As stated in Section 2, like D2B, the lengths of identifiers in Phoenix are not necessary the same, so that the dynamic operations are very low. Meanwhile, unlike D2B, our scheme can guarantee that each level of the trie tree maintains a complete deBruijn topology and the lengths of identifiers are the same. On the other hand, the degree of each peer is still a constant value. As a result, Phoenix achieve same routing efficiency as the static deBruijn graph does. In some extent, the following schemes can guarantee that Phoenix obtain the optimal tradeoff among autonomy, efficiency, robustness and load balancing.

The resource placement is very important to Phoenix. Normal resource placement policy may not organize resources and support resource queries accurately and efficiently. The following scheme can assure that resources are distributed based on the longest prefix matching policy. In addition, the scheme is fault-tolerant.

Algorithm 2 Placement(x)

```
1: if peer $x_1x_2...x_k$ has appeared in the overlay network
2:     The peer $x_1x_2...x_k$ stores the resource $x$.
3: else if its predecessor $y$ and $x'$ have common parent node in the tree
4:     peer $y$ is the host peer of the resource $x$.
5: else if its successor $z$ and $x'$ have common parent node in the tree
6:     peer $z$ is the host peer of the resource $x$.
7: else the youngest living ancestor of $x'$ is the host peer of resource $x$
```

All resources are kept by peers corresponding leaf nodes, and identifier of a resource is a 128 bits hash value. As shown in Fig. 1, resource 100... is stored by peer 100, and is taken over peer 101 when 100 is not alive. If peer 100 and 101 are both not alive, peer 10 keeps it.

In realty, d is often a large value for the sake of decreasing its diameter and improving its connectivity. d is usually adopted a number of tens, such as 20. The number of leaf peers is almost $d-1$ times than the number of inner peers. Suppose there is one million peers and d is 20. As a result, the number of leaf peers and inner peers are $950,000$ and $50,000$ respectively. Therefore, the leaf peers are enough to store all the resources. In addition, the scheme can guarantees the longest prefix matching policy. On the other hand, this design also provides a flexible algorithm for complex queries discussed in Subsection 5.7.

This resource placement strategy makes sure that any resource can be stored by an unified peer successfully even if its preferred and second host peers do not appear in Phoenix. On the other hand, this placement strategy makes Phoenix store any resource at a peer as close as possible to its preferred peer, and is the most important reference to lookup a resource.

The identifier of any peer or resource is acquired by using the MD5 algorithm on the IP and name of the node and resource. In the rest of this paper, we use the hash value to represent the original identifier.

5.2 Peer Joining

To ensure that our routing scheme proposed later executes correctly as a new peer participates Phoenix, the peer joining algorithm must ensure that the routing entries of each peer are up to date, and reduce the imbalance of the tire tree at low cost. If we want to keep a balanced tree, a new peer should join Phoenix level by level. If a level is full, a new peer may check direct leaf peer one by one to fix an empty leaf location. Otherwise, the tree may become unbalanced. In order to keep the characteristic of balance, the cost of the above process is high. Therefore the algorithm should reduce the level of imbalance in some extent at small cost.

To address the issue, we propose Algorithm 3. x is the hash value, and y is the contacted peer.

Algorithm 3 Join(x)

```
1: $x' = x_1x_2...x_k = \text{Route}(y, x)$
2: if peer $x_2...x_kx^i_{k+1}, 0 \le i \le d-1$, contains at least two identifiers and $x'$ is not the last unjoined child of $x_1x_2...x_k$ then
3:     $x'$ becomes $x_2...x_kx_{k+1}$ and Join($x'$).
4: else
5:     $a = \text{Predecessor}(x')$, $b = \text{Successor}(x')$.
6:     $x'$ connects to its parent $x'' = x_1x_2...x_{k-1}$
7:     Update $x'$'s $d$ out-neighbors in the deBruijn graph according to $x''$'s $d$ out-neighbors.
8:     if Comprefix($x'$, $a$) $= k-1$ and $a$ contains at least one identifier which is $\ge x'$ in the ring then
9:         $a$ transfers these identifiers to $x'$.
10:        Update all in-neighbors of $x'$.
11:    end if
12:    if Comprefix($x'$, $b$) $= k-1$ and $b$ contains at least one identifier which is $\le x'$ then
13:        $b$ transfers these identifiers to $x'$.
14:        Update all in-neighbors of $x'$.
15:    end if
16:    if $x'$ is the prefix of the identifiers contained in $x''$ then
17:        $x''$ transfers these identifiers to $x'$.
18:        Update all in-neighbors of $x'$.
19:    end if
20:    if $a$ contains data whose front $k$ string is $\ge x'$ in the ring then
21:        $a$ transfers the data to $x'$.
22:    end if
23:    if $a$ contains data whose front $k$ string is $\le x'$ in the ring then
24:        $b$ transfers the data to $x'$.
25:    end if
26:    $x''$ transfers all the data whose identifiers are the prefix of $x'$ to $x'$.
27: end if
```

Line 1 in Algorithm 3 indicates that a new peer x generates its identifier using MD5, and then contacts a peer y in Phoenix. Peer y routes the joining request of x to a suitable peer by the routing algorithm.

Lines 2-3 can reduce the level of imbalance. For example, as shown in Fig.1 (peer 100 and 101 fail and 010 connects to 10), let us assume a peer 0101 wants to become a child of 010, it finds that one neighbor peer 10 stores another string, then 0101 becomes 100 or 101 and it becomes a child of 10. Lines 5-19 are used to update the neighbors of a new peer. There is a special case that a peer contains at least one identifier. For example, the peer 10 transfers all the identifiers whose prefix is 100 to peer 100. For line 7, It is easy for us to have the following theorem.

Theorem 7. *For peer $x = x_1x_2...x_k$ and its d deBruijn neighbors $x_2...x_kx_{k+1}$, the deBruijn neighbors of all the offsprings of x are the offsprings of $x_2...x_kx_{k+1}$.*

From Theorem 7, we know x can find its deBruijn neighbors according to the neighbors of its parent at low cost. There is an easy way to implement the functionality of line 10 in the algorithm. Once x connects to a neighbor, it can find other neighbors along the predecessor link or successor link. If a neighbor does not exit, its sibling or father can replace it. The process of data movement is described in lines 20-27. At last, x truncates its 128 bit hash value to be the length of its level.

Theorem 8. *With the probability $1-O(1)$ the worst costs to handle a joining peer is $\lfloor\sqrt{d \log_d n} - 1.5 + \log_d((1+\varepsilon)\log n - O(\log\log n))\rfloor + 2\log_d n + o(1)$.*

The proof can be found in our technique report [26].

5.3 Peer Departure

P2P system is a highly dynamic network. Peers may leave and fail frequently. A robust scheme to handle the peer departure should be proposed. If a leaf peer with only one identifier wants to leave from Phoenix, it can leave freely. Its data is redistributed based on the data placement scheme. The behavior of peer departure should not have any impact on topology. If other peers want to leave, Algorithm 4 can repair the topology.

Algorithm 4 Departure(x)

```
1: $y$ = Predecessor($x$), $z$ = Successor($x$).
2: if Comprefix($x$, $y$) $= k-1$ then
3:   $y$ replaces $x$.
4:   All the children of $x$ connect to $y$.
5:   Update $x$'s out-neighbors in the deBruijn graph.
6: else
7:   if Comprefix($x$, $z$) $= k-1$ then
8:     $z$ replaces $x$.
9:     All the children of $x$ connect to $z$.
10:    Update $z$'s out-neighbors in the deBruijn graph.
11:  end if
12: else
13:   $x' = x_1x_2...x_{k-1}$ replaces $x$.
14:   All the children of $x$ connect to $x_1x_2...x_{k-1}$.
15:   Update $x'$'s out-neighbors in the deBruijn graph.
16: end if
Comprefix($x$, $y$)
Return the length of the longest common prefix of $x$ and $y$.
```

If a deBruijn in-neighbors of peer x, $x' = x_2...x_kx_{k+1}$, finds that its neighbor x' has leaved, x' just checks its routing table to find a candidate of peer x. Therefore the deBruijn in-neighbors of x do not need to update their routing tables. The cost is very low.

When an inner peer leaves, one of its neighbor replaces it. If many inner nodes fail, the existing neighbors may suffer from high load. Therefore, we should find a peer to share the load or replace the leaved peer. If an inner peer suffers from much load, it forwards a REPLACEMENT message to all offspring leaf peers. When a new peer joins one of the leaf peers. The leaf peer informs the new peer to share the load with inner peer, that is, replaces a leaved neighbor of the inner peer. If there is no new peer, the leaf peer which has only one identifier is used to share the load with the inner peer.

5.4 Peer Failure

The correctness and effectiveness of Phoenix relies on the fact that the $2d+1$ neighbors of each inner peer and $d+1$ neighbors of each leaf peer of are up to date.

To increase robustness and keep high effectiveness, each peer periodically detects those out-links. Since Phoenix is robust in the face of departure peers, once a peer is detected failed, the peer could be treated as a departure peer. The only difference is that a peer actively find a candidate for it before it leaves.

5.5 Load Balance

The peer joining algorithm can assure that a new peer chooses a smaller depth leaf as its father. Therefore, the scheme can reduce the level of imbalance of tire tree in a certain extent and the scheme cannot have an impact on the whole tree. Even if the tree is imbalanced, the peer joining, failure, departure and routing schemes can still assure that almost any data can be reached in an efficient way. Maybe some people consider that the management balance of the tree is not needed, and all the above algorithms are enough. But the more imbalanced the tree is, the higher the overhead of the algorithms are. In addition, the imbalance can lead to the skew distribution of data sets. Fig. 6 shows that peer c has over loaded. Hence, we still need a management scheme.

Thanks to our construction rule, some leaf peers can record the imbalance information. For example, as shown in Fig. 6, peer e, h, i, j, k, l and m have connections to peer c, and peer c records identifiers of their neighbors. The links from the low level to the high level are very important to our load balancing scheme, since they uncover the imbalance information of the tree.

To apply our load balance scheme, each peer keeps a data structure. When a peer constructs its routing table, it also records the IP information of its neighbor peers. The information is kept by the data structure. For example, c has kept the IP information of peer e, h, i, j, k, l and m. According to theorem 10, if peer c records the IP of e and c has no offsprings, peer c must record the IP of any offspring of peer e. We set an example to illustrate our load balance scheme. Peer c recorded all the information of the peers mentioned above, and can be regarded as a sever of these peers. In other words, c has a global view of the subtree e. When c is over load, it initiates the load balancing algorithm. There are two major steps of the load balancing algorithm. The first step aims at making the subtree e be balanced. The second step tries to guarantee that the leaf peers of the subtree e and c are at the same level. A rotation scheme in an AVL tree is used in the first step. Peer c sorts all the peers of the subtree e in an in-order traversal. The order is m, l, k, j, h, e, i. The rotation process is that m replaces l; l replaces k; k replaces j; j replaces h; h replaces e; and e becomes a child of i. Fig. 3(a) shows the result. With the same in-order traversal, the next result can be seen in Fig. 3(b). Finally, the subtree becomes balanced. Fig. 3(c) shows the final result. The second step is that m, k become the children of c and h, i become the children of m. Fig. 4 shows the result. After the algorithm is performed, the original subtree c and b are both balanced. The offsprings of c have shared load with it. The cost of data movement is low due to the rotation scheme. However, several peers change their position in the tree, and their routing tables are not up to date. The influenced peers are just a small part of the whole peers. It needs $O(\log n)$ hops to adjust routing table of a affected peer. Therefore, the load balancing algorithm do not result in large influence on the whole topology.

If too many overloaded peers perform the algorithm simultaneously with low probability, the overhead must be very high. To address this issue, we propose an delayed strategy to decrease the probability of executing the operations as low as possible.

From [20], we know the distribution of lifetime of each peer is followed Pareto distribution. Where,

$F(x) = 1 - (1 + x/\beta)^{-\alpha}, x > 0, a > 1$.

Therefore, we have

$E(L) = \beta/(\alpha - 1)$.

If a peer often becomes overload, it begins to execute the algorithm after $d^{k_0} \times E(L)$ time.

In addition, if the leaf peer in low level is overload and the peer does not connected to other lower peers, its parent or ancestors can share load with it.

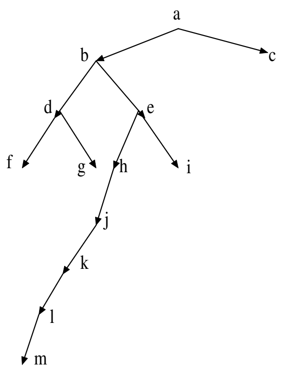

Figure 2: Example of a loaded peer(c)

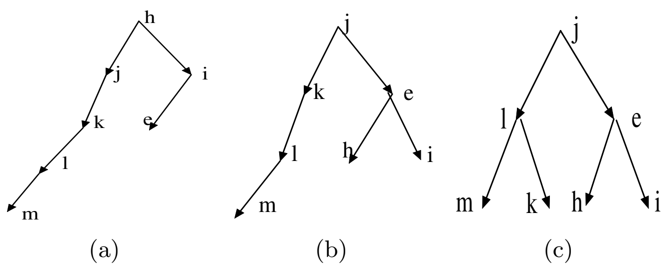

Figure 3: The process of the balancing for subtree e

To some extent, the algorithm can reduce the level of imbalance of the tire tree. The balance related management of the tree can be viewed as the following process:

An original Phoenix based on an initial d way complete trie tree can be constructed in advance. All leaf nodes are allocated to number of d^k peers, and the mapping is one to one. When many new peers participate in Phoenix, Phoenix may begin to become imbalance. Based on the peer joining scheme, a new peer often choose a leaf peer near to the root peer, and reduce the level of imbalance in a certain extent. If more and more new peers want to participate in Phoenix, the resulted overhead is high. Once the load balance algorithm is executed, the d trie tree can become balanced again. The process can be viewed as a expanding process of the topology. The peer departure process is viewed as a shrinking process of the expanding process.

Note that when the number of peers is small, the algorithm can help the tree become balanced quickly. When there are too many peers, it is not necessary to make the tree become balance again for the sake of avoiding huge overhead, fortunately our algorithms can keep high efficiency routing even under this scenario. In fact, the algorithm SHA-1 or MD5 used in most DHTs is the most important reason which leads to the imbalance of the system. The reason is that the identifers produced by SHA-1 are uniform in the 160 bits space. Therefore, for Phoenix, d should be large enough to reduce the level of imbalance of the trie tree. In practice, d is usually assigned a number of tens. We still use the MD5 method since the identifiers of the resource and node are computed locally according to their real names.

5.6 Robust and effective routing scheme

All the above schemes can assure Phoenix has a robust routing scheme. In this section, we will propose the routing algorithm.

The major messages handled by Phoenix can be partitioned as two categories, that is, messages to publish or query a resource, and messages to maintain the topology.

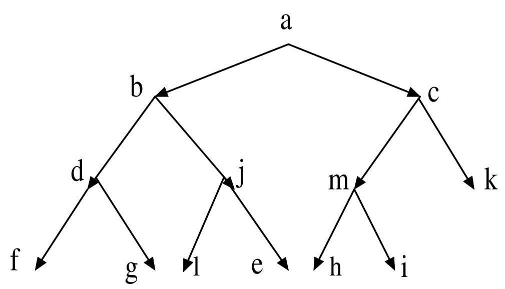

Figure 4: The result of the load balancing algorithm for a loaded peer)

Besides the above messages, Phoenix also should support the routing between any two peers. In order to route these kinds of messages to correct destinations effectively, each peer should keep a routing table and establish overlay connections to other at least $2d+1$ peers based to the topology construction rules mentioned above. But due to the high dynamic feature of P2P systems and delay of related topology maintenance algorithms, some connections may do not obey to the construction rules all the time. This fact requires the routing algorithm must be fault-tolerant.

We introduced a routing scheme in [19]. The short path from peer x to peer y as follows: find the largest suffix u of x that appears as a prefix of y, then walk towards a neighbor z of x such that its largest suffix v coincides with a prefix of y and the length of v is larger than that of u. These schemes work well in a stable environment, but suffer from poor robustness under dynamic environment like P2P network. The reason is that a message is always forwarded towards an unique neighbor that is more close to the destination than itself and other neighbors, and is not allowed to transfer along other paths.

To address all these issues, we propose a robust routing scheme, as shown in Algorithm 5. x is the requesting or the current peer, and y is the destination or the requested resource.

A query towards a resource y can be finished by two steps. First, it is forwarded to a peer which locates at same level as the beginning peer and has a common identifier prefix with y. Second, it is transmitted along down-link until the request reaches the leaf peer which keeps the y. For example, the peer wants to acquire a resource 111.... The request is forwarded along the link $00 \rightarrow 01 \rightarrow 11$, and then along down the link $11 \rightarrow 111$. A common feature of these scenarios is that all related peers are alive. If some related peers failed, the algorithm can also route successfully.

We give the following theorem to evaluate the performance of this routing algorithm.

Theorem 9. *In dynamic environment, the diameter of Phoenix is* $\lfloor \log_d n \rfloor$ *with the probability* $1 - o(1)$.

Note that a leaf peer may contain at least one identifier according to the construction rule. Algorithm 5 dose not work well for this kind of peers. To solve the issue, we proposed a improved algorithm as shown in our technical report.

5.7 Range Queries

In order to manage complex resources and support more wide applications, Phoenix should also support complex query operations besides exact-match query in a graceful fashion, for example, the single attribute range query, multidimen-

Algorithm 5 Route(x, y)

```
1: CASE1: Length($x$) = Length($y$) or ($y$ is a 128 bit hash value and $x$ is not a prefix $y$)
2: if $x = y$ then
3:   Return available.
4: else
5:   if Comprefix($x, y$) = $k - 1$ then
6:     if Comprefix($x$, Successor($x$)) = $k - 1$ and Successor($x$) is less than $y$ in the ring then
7:       Forward the message to peer Successor($x$).
8:     else
9:       if Comprefix($x$, Predecessor($x$)) = $k - 1$ and Predecessor($x$) is larger than $y$ in the ring then
10:        Forward the message to peer Predecessor($x$).
11:      end if
12:    else
13:      Forward message to peer $x$'s parent.
14:    end if
15:  end if
16: else
17:   if Exists at least one neighbor $z$ of x such that Common($z, y$) is larger than Common($x, y$) then
18:     Forward message to the peer which has the largest value of Common($z, y$).
19:   end if
20: end if
21: CASE2: Length($x$) < Length($y$) or ($y$ is a 128 bit hash value and $x$ is a prefix $y$)
22: if $y$ is a 128 bit hash value and $x$ contains $y$ then
23:   return available.
24: else
25:   if $x$ has at least one child then
26:     Forward message to its child $z$ which has the largest value of Comprefix($z, y$)
27:   end if
28: else
29:   $x$ forward message to its neighbors $z$ which has the largest value of Comprefix($z, y$)
30: end if
31: CASE3: Length($x$) > Length($y$)
32: if $x$ has a link to its parent then
33:   $x$ forwards message to its parent.
34: else
35:   $x$ forwards message to its neighbors $z$ which has the largest value of Comprefix($z, y$) other than $x$'s children.
36: end if
Common($x$, $y$)
Let $u$ be the longest suffix of $x$ which appears as a prefix of $y$.
Return the length of $u$.
```

sional data queries and multi-attributes queries. Here, we only discuss the queries for numerical values.

To handle the issues, the structure of Phoenix does not need to be changed. All the data is stored in leaf peers, and the inner peers act as the routing peers. The identifier of the peer is still produced by MD5. To support the order-preserving data placement strategy, the hash function for data needs to be changed. We recursively partition the attributes space of resources into sub-spaces in the same way to construct a complete trie tree whose height is $128 log_d 2$. Each data finds the smallest sub-space that contains its attribute values, and uses the identifier of that subspace as its identifier. Therefore, the data whose attribute values are close each other will obtain adjacent identifiers in the leaf peer level, and they will be stored on same or neighboring peers with the support of the order-preserving placement strategy. The identifier of the data is still 128 bit hash value.

To support the single attribute range query, the one dimension space is recursively partitioned as the above process. Each numerical value can find its corresponding hash value. Any peer that issues a range query can find the identifiers of smallest sub-space that contains the whole query region. Then the query covers multiply peers and will be routed towards the peer which charges the lower bound of it region firstly. Once the peer received the query, it will first forward it to the successor peer if it cannot cover the whole region of the query. And so on and so forth, the query will be forwarded to a peer which charges the upper bound of its region.

This method for range query can decrease the message cost caused by transferring the query to the all intersection peers, but the delay maybe be a litter larger than forwarding the query to all related peers simultaneously. It is needed to make a tradeoff between the delay and the message cost. After computing the hash values of the lower bound and upper bound, we get the common prefix y of the hash values. If they have no common prefix, we can divide the range into several (at most d) sub-regions with common prefixes and deal with each sub-region respectively. In addition, the parent of each leaf peer covers all its children's data region, and the parent also stores the regions of its children. And so on and so forth, every inner stores the region that its subtree covers, and each inner peer also stores its each child covered region. Then the request is routed to the peer a with identifier y. The results are covered by its offspring leaf peers of a. Finally, the request is routed along the down links of a and the request is routed to the peers which cover the result region. The request stops until the request reaches the leaf peers.

The two schemes for range query are flexible, if the traffic of the network is heavy, the first scheme is used. Otherwise the second scheme can be adopted.

For multi-attribute queries, we partition the entire multiple attribute space $< [l_0, h_0], ..., [l_i, h_i], ..., [l_{m-1}, h_{m-1}] >$ onto the completed trie tree along attributes $A_0, ..., A_i, ..., A_m$ in a round-robin style. Each peer in the tree represents a multiple-attribute subspace and the root node represents the entire multiple-attribute space. For any node B at the jth level of the trie tree that has d child peers, let i denote the value of $j \bmod m$. Then, the subspace c represented by node B is evenly divided into d pieces along the ith attribute, and each of its d child peers represents one such a piece. As a result, all leaf peer store the disconnected subspace of the entire space. Every attribute value $< a_0, ..., a_i, ..., a_{m-1} >$ can find its hash value.

In applications that involve multiple attributes, it is not uncommon for queries to involve only a small number of attributes (instead of all the attributes). Without loss generality, suppose part attributes $A_0, ..., A_i$ are often queried. The identifier of peer f which firstly partitions A_i can be computed based on hash function. If multi-attribute range queries are issued, the results are covered in the leaf peers of f. Each inner peer stores the region as dose for the single attribute range query. Then the request is routed to the leaf peers to fetch the results.

For multi-dimension data queries, all the schemes are the same as multi-attribute queries. The only difference is that all the dimensions $A_0, ..., A_i, ..., A_m$ are queried. The details are omitted because of the limitation of the space.

6. PERFORMANCE EVALUATION

We built a peer-to-peer simulator to evaluate the performance of our proposed system over large scale networks. We have carefully evaluated various aspects of Phoenix performance using the simulator. The simulation environment is windows XP, pentium 4 CPU 3.0 GHz, 1 G memory. For comparison purposes, we also implement Cycloid, Koorde, Viceory and BATON [21] in the simulator. For the fairness, the degree of each graph is $\log n$ for all the following experiments.

6.1 Routing Efficiency

In this section, we show the average path length of Phoenix is less than that of other network protocols. Queries are generated randomly and uniformly at each node, and the identifier of targeted resource are also uniformly distributed in the key-space. We evaluate networks consisted of 256 to 1 million nodes. The results are plotted in Fig.5. This figure shows that the delay of Phoenix and Koorde are the least one because they are both based on deBruijn. The delay of Phoenix is less than Koorde. The reason is that each node of Phoenix always has the even deBruijn connections for arbitrarily number of peers. Viceory gets the worst search delay, since each step only routes one level using the up and down links. Peers in each level of BATON are connected as Chord protocol. Therefore, it has a worse routing efficiency than other systems. Though Cycloid is similar as Viceory, it provides a more rapid routing in the decreasing and ascending phase. Cycloid provides a higher efficiency routing than Viceory.

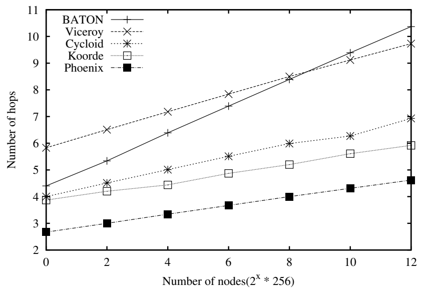

Figure 5: Average number of hops for a query

6.2 Range Queries

To evaluate the efficiency of range queries, CAN [1] and BATON [21] are chosen to compared with Phoenix. We also implement an improved range query scheme, called SPhoenix. Number of 4000 range query requests are uniformly distributed in entire network consisting of at lease 4000 and at most 1M peers. We calculate the average number of messages and hops induced by answering a rang query request by the network. As shown in Fig.7 and Fig.6, Phoenix results in least cost when performing a rang query request than CAN and BATON. This result is similar to that of the routing efficiency. This is because they both first identify a peer whose data is in the query region, then proceed left and/or right to cover the remainder of the query range. Though the parameter d of CAN is $1/2 \log n$, it still achieve the worst efficiency. In addition, it is very difficult to make decision on the value of d in advance, because it only relies on the expected number of nodes in the system, and therefore large value of d is not suitable in dynamic network. Compared to other systems, MPhoenix takes very small hops, but it pays much more price than others because the request can reach the leaf peers simultaneously.

To evaluate multi-attribute range query scheme, we suppose there are 20 attributes, and 12 of them are frequently used by queries. We called the algorithm MPhoenix. The average number of messages and hops of MPhoenix are recorded during experiments. In order to compare fairly, the range size is set same value among different range queries. The results of Phoenix and SPhoenix are plotted in Fig. 8 and Fig. 9. The figures show that the search efficiency of MPhoenix and SPhoenix are both great, but MPhoenix results in much cost.

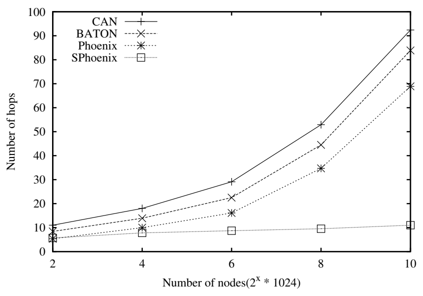

Figure 6: Average number of hops for range queries

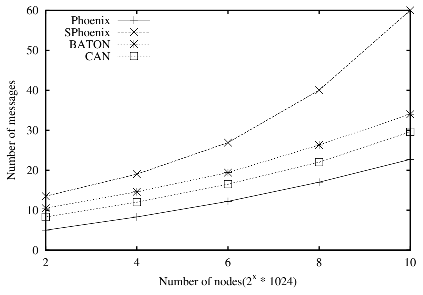

Figure 7: Average number of messages for range queries

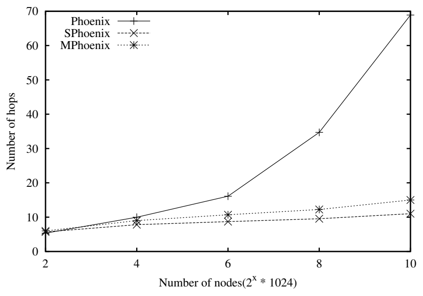

Figure 8: Average number of hops for multi-attribute range queries

6.3 Effects of Network Dynamics

In P2P systems, peers may join and leave at the same time. The level of intensity of nodes joining and leaving will have an exact impact on the network. We assume each system initially has 64k peers, and execute two experiments to evaluate the possible impact of dynamic operations. The first one is to evaluate the impact on routing efficiency. The second one aims at evaluating the average number of messages taken due to concurrent joining or leaving operations.

Fig.10 shows that the variation of average distance in the dynamic environment. The result confirms our claims that Phoenix is very stable compared with other systems when dealing with concurrent peer joining and leaving operations. The search delay of Viceory fluctuates most seriously because of its complex topology. When too many peers concurrently join and leave, the routing efficiency changes continually since topology becomes very unstable. Though the structure of Cycloid is also complex, the cost of a peer leaving operation is less than that of a peer joining operation, and therefore the search delay increase steadily. The path length of BATON is also stable due to the help of its good stabilization and load balancing scheme. In Koorde, the path length of a dynamic network changes little compared to that of a stable network. It is because that the first deBruijn node of each node and the deBruijn node's predecessors are updated in time.

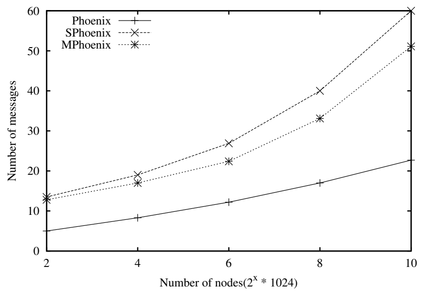

Figure 9: Average number of messages for multi-attribute range queries

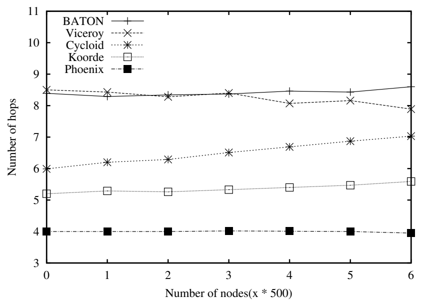

Figure 10: Average number of hops in the presence of peer joining and leaving

Fig.11 indicates the average cost of peer concurrent joining and leaving operations. The average messages required for routing the joining and leaving request, updating related routing tables and load balancing algorithm are all considered in our experiments. Koorde causes the highest cost, since it is based on Chord which requires $(\log n)^2$ messages for a peer joining or leaving operation. Compered with other systems, Phoenix pays a low price because it holds the most efficient routing. The leaving of a peer only takes $O(1)$ messages, and the neighbors need not update their routing tables. The total cost of a peer joining operation is $O(\log n)$. The load balancing algorithm only has an impact on a part of peers in Phoenix. Though BATON has lowest cost of a peer joining and leaving operations, the changes often frequently invoke the load balancing algorithm which takes more cost. BATON takes a higher cost than Phoenix. Cycloid and Viceory both take $O(\log n)$ messages to perform a peer joining or leaving operation, and therefore their costs are moderate among all the systems.

6.4 Load Balancing

A desirable feature of P2P systems is to distribute keys evenly among peers and make each peer receive similar number of requests forwarded by its neighbors. The percentage of peers whose load approximately equals to the average load is used to measure the load balancing ability of a P2P system. At most 5% error is allowed to evaluate the load balancing. For example, the average load is 1000 resources

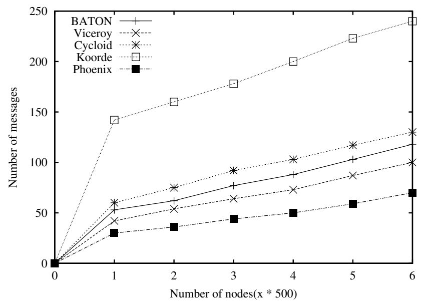

Figure 11: Average cost of peer concurrent joining and leaving

per peer, and all the values between 950 and 1050 are calculated in the percentage.

Number of 1000 peers are used in our experiment for each system. We varied the total number of resources from 10^4 to 10^5 in increments of 10^4. The results are plotted in Fig.12. As the increasing of number of resources, proportion of each system decreases. Phoenix and BATON are more stable and decrease least with the help of their load balancing algorithms. BATON however takes more cost. The number of resources per peer in Viceroy has a larger variation than other systems due to its unbalanced distribution. Cycloid leads to a more balanced resources distribution than Koorde. This is because the resources between a peers counter-clockwise neighbor and itself will be allocated to that neighbor or the node itself rather than to itself totally. There is no efficient load balancing algorithm for Cycloid. Cycloid has a poorer resources distribution than Phoenix and BATON.

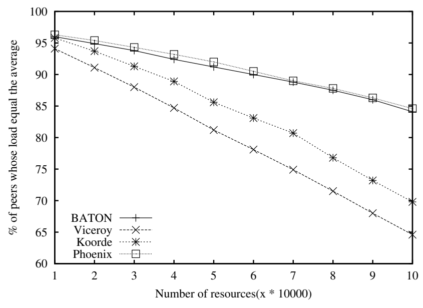

Figure 12: Variances of the resource distribution

6.5 Robustness

Number of 2000 queries are uniformly generated in the network consisting of 4000 peers. When the network becomes stable, we make each node fail with probability p ranging from 0.1 to 0.3. Fig.13 plots the percentage of successful queries when using the forwarding algorithm of corresponding protocol. For the same percentage of failed node, the percentage of successful queries for Phoenix is much higher than that of other systems, and the total variation of Phoenix is only a little. This is because all the schemes for Phoenix are enough robust to guarantee the successful query. BATON possess the worst fault tolerance capability, since the binary tree is prone to be not connected. In addition, once the adjacent link connecting the high and low level peers fails, the routing will be influenced greatly. There are not many differences for robustness between Cycloid, Viceory and Koorde, because they almost adopt the same fault tolerant scheme.

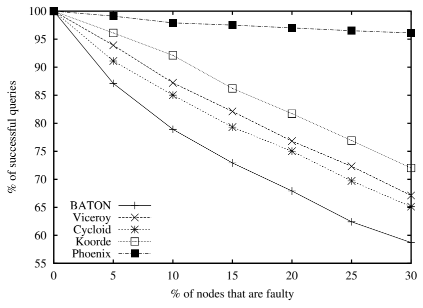

Figure 13: Percentage of successful queries

7. CONCLUSIONS

Firstly, the lower bounds of network diameter and average routing distance of P2P systems are given in high dynamic environment. In Section 4, the d-way trie tree is proposed as a universal method to support any static network, and then design Phoenix, an efficient and practical P2P architecture. In Phoenix, the order-preserving hash scheme and resource placement strategy are employed to support both exact and complex queries. The topology management and routing schemes guarantee flexible resources distribution and robust topology. Phoenix achieves optimal diameter, high performance, and good connectivity for dynamic P2P networks. In the future, we will use Phoenix as an infrastructure to support large scale distributed applications.

8. REFERENCES

[1] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In Proceedings of the ACM SIGCOMM, 2001.

[2] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object ocation and routing for large-scale peer-to-peer systems. IFIP/ACM Int. Conf. Distributed Systems Platforms, Heidelberg, Germany, 2001.

[3] I. Stoica, R. Morris, D. Karger, et al. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. ACM SIGCOMM, 2001.

[4] B. Y. Zhao, J. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. SIGCOMM, 2001.

[5] S. Q. Zhuang, B. Y. Zhao, and A. D. Joseph. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In ACM NOSSDAV, 2001, pp.11-20.

[6] P. Fraigniaud and P. Gauron. The content-addressable network d2b. Technical Report Technical Report LRI 1349, Univ. Paris-Sud, 2003.

[7] F. Kaashoek and D. R. Karger. Koorde: A simple degree-optimal hash table. In 2nd International Peer-To-Peer Systems Workshop (IPTPS), 2003.

[8] D. Loguinov, A. Kumar, V. Rai, and S. Ganesh. Graph-Theoretic Analysis of Structured Peer-to-Peer Systems: Routing Distances and Fault Resilience. Texas A&M Technical Report, 2003.

[9] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. ACM Symp. on Principles of Distributed Computing, 2002.

[10] M. Naor and U.Wieder. Novel architectures for P2P applications: The continuous-discrete approach. In ACM SPAA, Jun. 2003, pp.5-59.

[11] J. Xu, A. Kumar, and X. Yu. On the fundamental tradeoffs between routing table size and network diameter in peer-to-peer networks. IEEE J. Sel. Areas Commun., vol. 22, no. 1, pp. 15-163, Jan. 2004.

[12] N. Harvey, M. Jones, S. Saroiu, et al. Skipnet: A scalable overlay network with practical locality properties. The 4th USENIX Symp. on Internet Technologies and Systems, Washington, USA.

[13] H. Shen, C. Xu, and G. Chen. Cycloid: A constant-degree and lookupefficient p2p overlay network. The 18th Int. Parallel and Distributed Processing Symposium, New Mexico, USA, 2004.

[14] S. Banerjee and D. Sarkar. Hypercube connected rings: a scalable and fault-tolerant logical topology for optical networks. Comp. comm., 24:106-1079, 2001.

[15] D. Li, X. Lu, and J. Wu. Fissione: A scalable constant degree and low congestion dht scheme based on kautz graphs. INFOCOM, 2005, pp.167-1688.

[16] X. Wang, D. Loguinov. Load-Balancing performance of consistent hashing: asymptotic analysis of random node join. In IEEE/ACM Trans. on Networking, 2007.

[17] W. Szpankowski. Patricia Trees Revisited Again. Journal of the ACM, vol. 37, 1990.

[18] M.H. DeGroot. Probability and Statistics. Addison-Wesley, 2001.

[19] D. Guo, J. Wu, H. Chen, and X. Luo. Moore: An extendable peer-topeer network based on incomplete kautz digraph with constant degree. In Proc. 26th IEEE INFOCOM, May 2007.

[20] F.E. Bustemante and Y. Qiao. Friendships that last: peer lifespan and its role in P2P protocols. International Workshop on Web Caching and Distribution, September 2003.

[21] H. V. Jagadish, B. C. Ooi, and Q. H. Vu. BATON: A balanced tree structure for peer-to-peer networks. In Proceedings of the 31st VLDB Conference, 2005.

[22] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: supporting scalable multi-attribute range queries. SIGCOMM Comput. Commun. Rev., 34(4), 2004.

[23] H. V. Jagadish, B. C. Ooi, Q. H. Vu, eq al. VBI-Tree: A peer-to-peer framework for supporting multi-dimensional indexing schemes. ICDE, 2006.

[24] H. Jagadish, B. C. Ooi, K.-L. Tan, Q. H. Vu, and R. Zhang. Speeding up search in peer-to-peer networks with a multi-way tree structure. In SIGMOD, 2006.

[25] A. Crainiceanu, P. Linga, A. Machanavajjhala. P-Ring: An efficient and robust P2P range index structure. In SIGMOD, 2007.

[26] Y. Yuan, D. Guo, G. Wang, Y. Liu. A unified construction method of P2P networks with lower bound to support complex queries. HKUST Technical Report, 2008. http://www.cse.ust.hk/ liu/guodeke/index.html.

[27] Y. Chawathe, S. Ramabhadran, S. Ratnasamy, et al. A case study in building layered dht applications. In SIGCOMM, 2005.

[28] C. Zheng, G. Shen, S. Li, and S. Shenker. Distributed segment tree: support of range query and cover query over DHT. In IPTPS 2006.

[29] E. Tanin, A. Harwood, and H. Samet. Using a distributed quadtree index in peer-to-peer networks. VLDB Journal, 16(2):16-178, January 2007.

[30] M. Cai, M. R. Frank, J. Chen, and P. A. Szekely. Maan: A multiattribute addressable network for grid information services. In GRID, 2003.