Available online at www.sciencedirect.com
ScienceDirect
Computer Networks 52 (2008) 1451–1472
Computer Networks
www.elsevier.com/locate/comnet

Approaching neighbor proximity and load balance for range query in P2P networks ☆

Yuh-Jzer Joung *

Department of Information Management, National Taiwan University, No. 1, Sec. 4, Roosevelt Road, Taipei 10617, Taiwan, ROC

Received 25 October 2006; received in revised form 3 January 2008; accepted 28 January 2008
Available online 12 February 2008

Responsible Editor: K. Kant

Abstract

We present a platform for building a range-queriable system. A distinguishing characteristic of the platform is that objects are not bound to any particular node. Rather, they can be freely moved between neighboring nodes to balance load as long as some ordering between objects in different nodes is respected. Decoupling the link between nodes and objects allows the overlay network to be constructed independently of object placement, focusing only on its own concerns, e.g. proximity and balanced load in routing. The main focus then is to present several heuristics for proximity-aware node joining to reduce communication costs in the platform. These heuristics are based on a novel process called *hierarchical neighborhood search*, which utilizes the overlay links to provide incoming nodes an overview of the network topology, thereby to guide them to find their neighborhood in the overlay. The experimental results show that some of the heuristics are very effective; they can reduce as much as 80% of the communication costs for neighboring nodes, and 40% overall.

Keywords: Peer-to-peer (P2P); Range query; Proximity-aware node join; Load balance; Chordal graph; Hierarchical neighborhood search

1. Introduction

A fundamental problem in peer-to-peer (P2P) applications is to search and locate a desired object in the network. With the help of some centralized servers, e.g. Napster, object information can be gathered at the servers to provide efficient lookup for the peers. Moreover, a variety of query formats (e.g. prefix match and range query) can be easily supported as they were in a centralized information system. However, the centralized servers are vulnerable to hot spots and single point of failures.

Search in a fully decentralized P2P overlay is somewhat challenging. There are two paradigms: one leaves the overlay *unstructured*, and locates a desired object basically by flooding query messages to the network. The other maintains the overlay as a *distributed hash table (DHT)* that allows objects to be efficiently located via their keys.

The advantages of unstructured overlays are that they can be maintained in a very casual style; there

☆ Research supported in part by the National Science Council, Taipei, Taiwan, Grants NSC 93-2213-E-002-096, NSC 94-2213-E-002-036, and NSC 95-2221-E-002-058.
* Corresponding author. Tel.: +886 2 33661183; fax: +886 2 23627094.
E-mail address: joung@ntu.edu.tw

1389-1286/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.comnet.2008.01.016

is not much restriction on which nodes should be neighboring to each other. Several practical overlays have been built in the Internet environment (e.g. Gnutella and Freenet [1]), and empirical studies [2] have shown that they are very stable even in a highly dynamic environment. The main problem with them is that search cost is relatively high, as query messages are blindly passed one node after another to check whether they have the desired objects. Although much work has been devoted to improving their search performance (e.g. [3–7]), search can still not be guaranteed, meaning that an object may not be located if it is too far from the querying node, unless the entire network is to be searched out.

DHTs (e.g. Tapestry [8], Pastry [9], Chord [10], and CAN [11]), on the other hand, guarantee search results under reasonable costs. This is because nodes in the overlays maintain their routing tables in a way that guarantees a path from any given source to a destination, and the path length is usually bounded by $O(\log N)$ hops, given only $O(\log N)$ entries in each routing table, where N is the network size. If the node hosting an object is known, then locating the object becomes simply a message routing from the querying node to the destination target.

To map objects to nodes, we can use object names (or attributes) as keys. Object names are often human-readable, e.g. "Lord of the rings-I" and "HP LJ2200d", and so can be ordered, say, alphabetically. This ordering allows us to infer some correlation between objects, for example, "HP LJ1200", "HP LJ2200", and "HP LJ5500" are a family of LaserJet printers, and "Lord of the rings-I.mpg", "Lord of the rings-II.mpg", and "Lord of the rings-III.mpg" are a series of movies. *Prefix search* or *range query* can therefore be built based on this ordering to allow users to enter a more flexible query such as "Lord of the rings*" to retrieve the series.

Depending on how the order of object names is preserved, there are three ways to map object names to nodes: *linear*, *hash*, and *even and order-preserving* (see Fig. 1). Linear mapping preserves the order of object names. Range query in DHTs then corresponds to locating a set of nodes whose IDs fall within some interval. Many DHTs (e.g. Tapestry, Pastry, Chord and CAN) use routing schemes akin to prefix/suffix matching, and so can facilitate range query. However, objects typically do not stay uniform in the original name space. Thus, linear mapping causes uneven distribution of objects.

A common way to cope with load balancing is to hash object names to identifiers that are uniformly distributed over an ID space; see Fig. 1b. Then, by a simple mapping from object IDs to node IDs, the load of nodes can be evenly distributed, thereby avoiding hot spots and enhancing stability. However, object ordering is lost after hash, and exact name must be entered to locate an object; a slight difference in a query causes totally irrelevant results. It is a pity that exact name match does not facilitate a friendly human query. Some systems (e.g. [12]) impose an additional data overlay by attaching objects with pointers to preserve the connectivity between neighbors in the original name space. Unless additional routing information is built, the cost is high to retrieve successive objects, as each link in the data overlay corresponds to a routing path between two nodes in the DHT overlay, which costs $O(\log N)$ messages.

A more sophisticated way to deal with range query is to design an order-preserving mapping from object names to ID space that can also ensure a uniform distribution; see Fig. 1c. For example, a

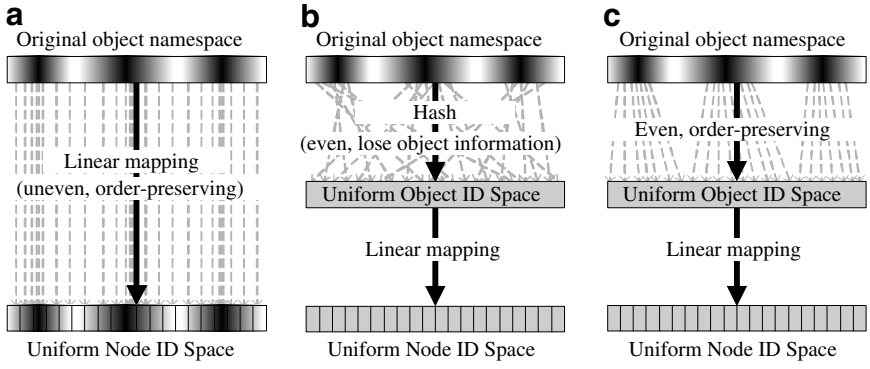

Fig. 1. Three mappings from objects to nodes.

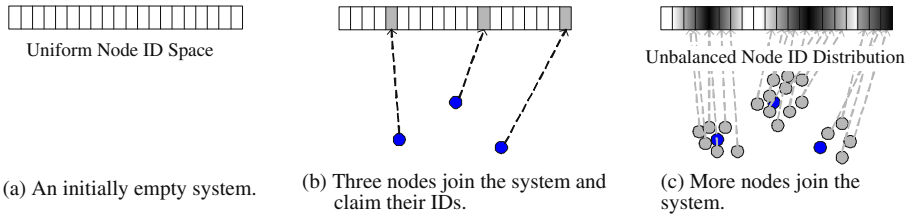

Fig. 2. Proximity results in an unbalanced node ID distribution.

statistic database is used in [13] to translate object names into binary strings that can still preserve the order of the names. The binary strings are then used as identifiers to determine the hosting nodes of objects. As one can see, some global knowledge on the distribution of object names is needed to design an even, order-preserving mapping from object names to an ID space. This also makes an implementation too dependent on the object set.

Load-balanced range query is complicated by considering proximity in the overlay. Proximity, in general, reduces communication costs in the overlay, and is particularly important in range query because the search process involves many activities between neighboring nodes (with respect to the ordering of objects). However, it is not easy to cope with both proximity and load balancing in a DHT overlay. This is because the assignment of node IDs in the overlays is usually enforced by some rules, e.g. an incoming node has its ID fall in between its neighboring nodes. As a result, the distribution of node IDs (whose nodes are physically present) may not be uniform. For example, in Fig. 2, suppose three nodes have joined a system and, subsequently, nodes in their vicinity also wish to join. Since the potential IDs of the new nodes are restricted within the range close to the existing ones, the final occupied IDs will yield an unbalanced distribution. Hence, proximity might dilute the aforementioned efforts made to balance loads.

In this paper we take load balancing and proximity into account, and present a platform, called *chordal graph*, for efficient range query. Like DHTs, search is guaranteed with reasonable costs. Unlike DHTs, however, objects are not bound to any particular node. Rather, they can reside at any node as long as some ordering between objects in different nodes is respected. Thus, objects can be freely moved between nodes to balance load. Our focus is to present several heuristics for proximity-aware node joining to reduce communication costs in the platform. These heuristics are based on a novel process called *hierarchical neighborhood search*, which utilizes the overlay links to provide incoming nodes an overview of the network topology, thereby to guide them to find their neighborhood in the overlay. Note that no global knowledge is required in guiding the join process, or in distributing objects. The experimental results show that some of the heuristics are very effective; they can reduce as much as 80% of the communication costs for neighboring nodes, and 40% overall.

2. Related work

Recently there has been a flurry of research on range query in structured P2P networks. First, we note that a fundamental issue for range query is to be able to efficiently traverse in between nodes whose key ranges are adjacent. Some DHTs, e.g. Chord, has this property in nature. Then, a simple way to support range query is to use a locality preserving hash function to map object keys (which are semantically meaningful) to the Chord ID space [14]. For DHTs that do not have the key range adjacent feature, e.g. CAN, Space Filling Curves have been applied [15]. In addition to DHTs, distributed data structures suitable for range query have also been explored, e.g. tries [16,17], SkipNet [18], and skip graph [19,12,20]. Moreover, as semantically meaningful key space is often skewed in key distribution, load balancing has also been explicitly addressed for range queriable systems, e.g. [21–23].

We want to re-emphasize that a network-aware system is beneficial to range query as neighboring communication occurs heavily both in adjacent key retrieval and in load balancing. System communication costs can be greatly reduced if we can bring neighboring nodes in physical network close in the overlay. However, locality has not been explicitly and fully addressed for range query, but is often considered independently during, e.g. overlay construction [24–27], routing table entry selection

[28,29], route selection [28], and periodical maintenance [30,31].

In fact, network locality has also been considered in SkipNet [18]. SkipNet organizes nodes and data objects according to their lexicographic addresses that reflect organization boundary. Load balance, however, is somewhat compromised as load can only be balanced within each organization. Since load balancing and node proximity may conflict with each other, it is useful to take both issues into account when designing a range queriable system. In the next section we present our platform, and introduce a very simple scheme to balance loads. Then in Section 4 we take proximity into account and present several heuristics for proximity-aware node joining. Section 5 presents a comprehensive study of the heuristics. Conclusions and future work are offered in Section 6.

3. System model

In this section we present a general platform for range query. We assume that each object o has a unique key $key(o)$. When no confusion is possible, we simply use o to represent its key to simplify the notation. Keys are totally ordered by '$\leqslant$', and may bear some semantic meaning. Objects can be retrieved by the following two basic functions:

- $search(k)$: return the object associated with key k.
- $range\_Query(k_1, k_2)$: return the objects whose keys fall in between k_1 and k_2.

More flexible queries can be built from range query. For example, the prefix query "abc*" can be translated into a query of range in between "abc" and "abd" (exclusive).

3.1. A platform for range query

To enable the above two search functions in a distributed environment, we can partition the key space into several ranges and assign each range to a node in the system. For example, in Fig. 3, the object name space $[A..Z]$ is partitioned into five intervals, each handled by a node, such as $[J..L]$ for node 3. For a minimum operation of the system, each node u maintains a link $u.next$ to its successor. Since the intervals are ordered as a circular list, search is simple: on receiving a request for some object, if the object key falls within the node's responsible range, then the node answers the query by checking whether it has the object; otherwise, it forwards the request to its successor. Note that search is guaranteed, and range query is as simple as the basic search. Also note that unlike DHTs, node IDs are not used in search. As long as some ordering among objects in the list is maintained, there is no restriction on where an object should be placed. Furthermore, since node ID is not used in the search process, nodes can be freely positioned in the list, thus leaving much room for incorporating network proximity into the design.

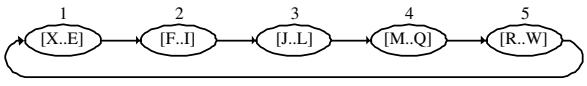

Fig. 3. A circular list for search.

For efficiency, we add more links to speed up the search process. Each node u maintains a number of links such that higher level links connect to farther nodes than lower level links. We use $u.next[i]$ to denote the node connected to by the level-i link, $i \geqslant 0$. See Fig. 4 for an illustration. The link structure resembles a Chord ring and so we will also refer to it as a *chordal graph*. For a distinguishing purpose, we sometimes refer to level-i links, where $i > 0$, as *express links*, while level-0 links as *base links*. By definition, base links in a chordal graph induce a circular list. We refer to the list as the *base list* (or *base ring*) of the chordal graph, and use C_B to denote a chordal graph C with base list B. The *size* of B, denoted by $|B|$, is the number of nodes in the list. There are several ways to maintain an overlay of the chordal graph structure, e.g. SkipNet [18,32], skip graph [19], and Chord [10]. For our purpose here, there are subtle differences among them. We will return to this issue later in Section 4.3.

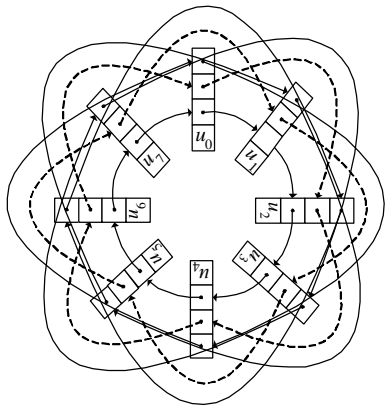

Fig. 4. A chordal graph.

In the rest of the paper, unless stated otherwise, we shall assume that each node in a chordal graph C_B is responsible for a key space range, and the key space ranges together cover the entire key space. Moreover, let $[u.lower, u.upper)$ denote u's key space range. Then the key space ranges are *circularly ordered* such that $u.upper = u.next[0].lower$. Note that there can be at most one node having a lower bound larger than the upper bound (e.g. node 1 in Fig. 3). This node is referred to as the "turning node". Every object handled by a node u must have its key fall within u's range. Therefore, objects maintained by the nodes in B are also "circularly ordered".

The following algorithm gives a basic search procedure on chordal graphs. The idea is to use higher links whenever possible without "overshooting". In the algorithm, *u.height* denotes the maximum level of links maintained at u.

Algorithm *u.search(key)*
1. **if** $key \in [u.lower, u.upper)$
2. **if** *key* exists in the set of objects hosted by u
3. **return** object with the key
4. **else**
5. **return** NOT_FOUND
6. **else** {
7. **for** $i = u.height$ **downto** 0
8. **if** $key \in [u.next[i].lower, u.lower)$ {
9. $u.next[i].search(key)$// forward search message to $u.next[i]$
10. **return**}
11. (* else down one more level *)
12. **return** NOT_FOUND}

As we can see, if each node maintains $\log|B|$ links and the level-i link connects to the 2^ith successor in the base list, $0 \leqslant i \leqslant \log|B| - 1$, then it takes at most $O(\log|B|)$ forwardings to locate an object. Existing techniques (e.g. [10,18,32,19]) are able to maintain a chordal graph of this property. Insert, delete, and update of objects can be easily built from the basic search. Moreover, since objects in a chordal graph are circularly ordered, range query can also be easily performed based on the above search procedure. For example, the function $range\_Query(k_1, k_2)$ can be implemented by first locating the node handling k_1 using the basic search function, and then passing the query message from the node along the base list until the node handling k_2 is reached.

The above implementation visits nodes responsible for a key range sequentially. If the range is large, one may also design a parallel search scheme to speed up the process. The idea is to partition the query range so that each subrange can be searched independently. To illustrate, consider Fig. 5. Suppose u receives a query of range $[a, b]$, where $a \in [u.next[1].lower, u.next[2].lower)$ and $b \in [u.next[4].lower, u.next[5].lower)$. Then u can partition the range $[a, b]$ into the following subranges $[a, u.next[2].lower)$, $[u.next[2].lower, u.next[3].lower)$, $[u.next[3].lower, u.next[4].lower)$, and $[u.next[4].lower, b]$, and issues four queries simultaneously to $u.next[1]$, $u.next[2]$, $u.next[3]$, and $u.next[4]$ to search the four subranges, respectively. Each of the four nodes, upon receiving the query from u, uses the scheme recursively to search objects in the assigned range.

3.2. A simple distributed storage balancing scheme

The main problem with the above search platform is that object keys are often not distributed evenly in the key space, particularly when the keys carry some semantics. Without the knowledge of the actual key distribution, an arbitrary partition of the potential key space may result in an unbalanced load distribution where some nodes host significantly more objects than the others. However, observe that the key space ranges of two neighboring nodes can be flexibly adjusted without affecting the circular ordering in the base list. This means that objects can be "freely drifted" between neighboring nodes. More precisely, if a node u finds that its load ratio over its successor v exceeds some threshold, then u may move its objects within the range $[j, u.upper)$ to v, and adjusts their key space ranges to $[u.lower, j)$ and $[j, v.upper)$, respectively. The value j is chosen to balance the load. We refer to this scheme as *free drift*.

Free drift can be performed distributedly in quite a casual style to balance load [22]. When the operation is performed distributedly, the combined effect is that the overall load distribution among nodes can be well balanced. Another situation for free drift is when a node is to leave or join a list, in which it can either obtain some share of objects from its

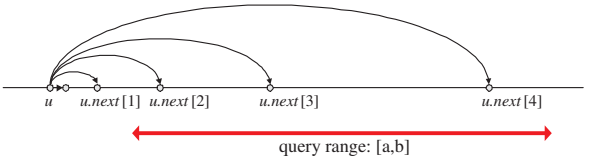

Fig. 5. Partition of query range for parallel search.

neighbors (upon joining the list), or distribute its objects to them (before leaving the list).

4. Approaching neighbor proximity

As the membership of nodes and the amount of objects change, there will be communications between neighboring nodes in the base list due to free drift. Range query also involves lots of communications in the base list as search messages are hopped through a sequence of neighboring nodes. Therefore, the system performance can be increased by decreasing the neighboring communication cost. Since communication in the overlay must go through the underlying physical links, building a network-aware overlay can help reducing the communication cost. There may be schemes available to optimize an overlay layout given a physical topology. Considering the size and the dynamics of a P2P network, however, it is infeasible to employ such an optimization scheme. Therefore, heuristics will be used to offer a good overlay construction. Below we first define some measures for a chordal graph. Then we present several heuristics for proximity-aware overlay construction.

4.1. Measures of chordal graphs

We consider two measures for chordal graphs, one concerning the communication cost of free drift, and the other concerning the cost of query. We assume a distance function d for measuring communication cost between two nodes u and v. For simplicity, d is static and symmetric, i.e., $d(u,v) = d(v,u)$.

Let C_B be a chordal graph with base list $B = u_0u_1\cdots u_{|B|-1}$. Since free drift occurs only in neighboring nodes, the communication cost of a free drift (of a unit object) between u_i and u_{i+1} is in linear proportion to $d(u_i,u_{i+1})$ (addition and subtraction on node indices are to be interpreted modulo $|B|$). Suppose free drift occurs with the same probability between any two neighboring nodes in B. Then the cost of a free drift in C_B can be defined as $\mathrm{FD}(C_B)$, where

$$\mathrm{FD}(C_B) = \frac{\sum_{i=0}^{|B|-1} d(u_i,u_{i+1})}{|B|} \tag{1}$$

That is, $\mathrm{FD}(C_B)$ is the average link distance in the base list.

To measure the communication cost of a query, consider Algorithm *search* in Section 3.1. The execution of the algorithm to resolve a query Q involves some overlay communications to forward the query (due to line 9). Let $F_Q = \{e_1,\ldots,e_k\}$ be the set of overlay links that are traversed. The communication cost of Q can be measured by $\sum_{i=1}^{k} d(e_i)$. To calculate $\sum_{i=1}^{k} d(e_i)$, assume that a query has equal probability to originate at any node u_i, and has equal probability to search object in any other node u_j in the base list. Moreover, for simplicity, assume that C_B is a "perfect" chordal graph such that each node u_i maintains $\log|B|$ links where the level-j link connects to the 2^jth successor; i.e., $u_i.next[j] = u_{i+2^j}$, $0 \leqslant j \leqslant \log|B| - 1$. We make the following three propositions:

Proposition 4.1. *F_Q is composed of at most $\log(|B|)$ links, and all the links have strictly decreasing levels.*

Proposition 4.2. *For each link level in C_B, the probability is $1/2$ that the level has a link in F_Q.*

Proposition 4.3. *For each link level in C_B, all the $|B|$ links at the level have equal probability to be included in F_Q.*

Proposition 4.1 follows directly from the search algorithm. Proposition 4.2 follows from the fact that when the search proceeds to level j at node u_i, the link $(u_i, u_i.next[j])$ will be used to forward the query message if the target object falls within the range $[u_i.next[j], u_{i+2\cdot 2^j})$ in the base list. Note that the search range at this level is $[u_i, u_{i+2\cdot 2^j})$, and $u_i.next[j] = u_{i+\cdot 2^j}$. Since we assume that the target object has equal probability to fall within any node in the base list, the probability is $1/2$ that a link at level j will be used by Q. Proposition 4.3 follows from the fact that Q has equal chance to originate from any node in the base list.

From the three propositions, we see that the routing cost of a random query Q in C_B is

$$\begin{aligned}
&\mathrm{RT}(C_B)\\
&= \frac{1}{2}\sum_{0\leqslant h\leqslant \log|B|-1} \frac{\text{total length of level} - h \text{ links}}{|B|}\\
&= \frac{1}{2}\,\frac{\text{total length of all links in } C_B}{|B|}\\
&= \frac{1}{2}\log|B| \times \frac{\text{total length of all links in } C_B}{\log|B| \times |B|}\\
&= \frac{1}{2}\log|B| \times \text{average link length in } C_B
\end{aligned} \tag{2}$$

In the above we consider only query of a given key. For range query, both FD and RT affect the cost of range query, as the search process involves a message routing from the querying node to a node whose keys are in the search range, and then a sequential search in its neighborhood.

4.2. Some heuristics for proximity-aware node joining

In this section we propose some heuristics for node joining that make use of the chordal graph structure to improve the overlay proximity. For intuition of these heuristics, observe that express links in a chordal graph provide a node an overview of the object layout to quickly narrow down the search scope. For example, consider Fig. 6. When a node x receives a query, it starts with the highest level link to see if the search scope is in the range $[x..z)$, where it should continuously process the query using the next level link, or beyond z, where it should forward the query to z. Similarly, if z takes over the query processing, it will use its next level link to see if the search scope is within $[z..w)$, or beyond w, and then takes an appropriate action.

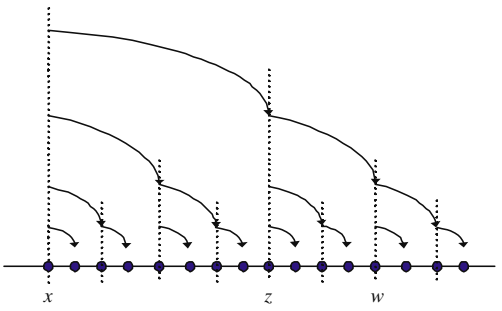

Fig. 6. A search tree in chordal graphs.

Before continuing the discussion, we first note that so far we have considered chordal graphs with only forward links. It is easy to see that if we maintain links in both forward and backward directions, then the expected search cost can be reduced (at the price of doubling the link maintenance cost). Although only one direction is necessary, maintaining both forward and backward links at each level also allows our proximity-aware node joining algorithms to be easier to present. Therefore, in what follows we shall assume that the underlying chordal graph is doubly-linked, and use $u.next[i]$ and $u.prev[i]$, respectively, to denote u's *forward* and *backward* neighbors (or, alternatively, *successor* and *predecessor*) at level i.

As commented above, express links provide nodes a quick overview of the object layout. If the overlay reflects network proximity, then express links also provide incoming nodes an estimate of the network layout to allow them to quickly locate their neighborhood. This is because higher level links will point to farther nodes (see the link length pattern in Section 5.5.1). Therefore, when a new node v joins the system via some contact node x, a natural approach to find v's spot in the overlay is to use x as *pivot*, and starting from x's highest express links, checks whether v should be placed close to x's successor, to x's predecessor, or to x itself. See Fig. 7. Once the area is chosen, the neighborhood search process can be recursively applied by using one of the three nodes (x and its two neighbors, depending on which one is closer to v) as new pivot to narrow down the search area until the ground level is reached. We call this a *hierarchical neighborhood search process*, as it starts from some highest level links to overview the network topology, and then gradually follows the link level to narrow the search scope until an incoming node's

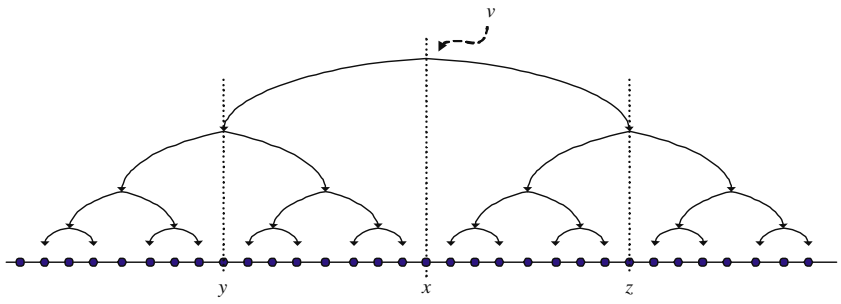

Fig. 7. A hierarchical neighborhood search process: an incoming node v first uses x as pivot to see if it is closer to x, to y, or to z; and then recursively uses the closest node as new pivot to compare with its next level neighbors.

neighborhood is located. In Fig. 7, the nodes that can be potentially used as pivots to search for a new node's position form a tree. We will refer to the tree as a *locality guidance tree (LGT)*.

In the above we need a function to measure the closeness of a node v to a target p. We use $v.Affinity(p)$ to denote this function. The following three algorithms F0, F1, and F2 are of different concerns. In F0, the closeness is measured directly by the distance between v and p. In F1 and F2, the closeness is measured by the average distance between v to p's "neighborhood". In F1, the neighborhood concerns only p's two neighbors in the base list. In F2, we take express links into account and assign weights to them. Two different weighting schemes can be considered. For the *uniform* weighting scheme ($\Delta = 1$ in the algorithm code), the links are of equal importance. For the *upward-decreasing* scheme ($\Delta = 2^{-i}$), the weights are $\frac{1}{2}, \frac{1}{4}, \ldots, \frac{1}{2^i}$ from the lowest level upward to the level in consideration.

Algorithm (F0). $v.Affinity$ (node p)

(* Closeness is measured directly by the distance between v and p. *)

1. **return** $d(v,p)$

Algorithm (F1). $v.Affinity$ (node p)

(* Measure the average distance of v to p's two neighbors in the base list. *)

1. **return** $(d(v,p.prev[0]) + d(v,p.next[0]))/2$

Algorithm (F2$^{\Delta}$). $v.Affinity$ (node p, level h)

(* Δ can be 1 or 2^{-i}, and assume that p is not isolated *)

```
1.    dist := 0
2.    weight := 0
3.    for i := h downto 0
4.      if p.prev[i] exists
5.        dist := dist + d(v, p.prev[i]) * Δ
6.        weight := weight + Δ
7.      if p.next[i] exists
8.        dist := dist + d(v, p.next[i]) * Δ
9.        weight := weight + Δ
10.   return (dist/weight)
```

Note that in a node join process, we may also take a more greedy approach by allowing a new node to replace an existing node at its position. For example, in Fig. 7, at any point in the join process, if a new node v finds that its placement at some position can bring better performance than the old one does at the same position, it replaces the old node. The replaced node then "re-inserts" itself into the system using the same join process. We refer to such option as *role exchange*. The closeness measurements F1 and F2 are designed to facilitate role exchange, as $v.Affinity(p)$ represents how close v is to p's neighborhood. Thus it is meaningful to compare $v.Affinity(p)$ and $p.Affinity(p)$.

Observe that a chordal graph C_B is cyclic in structure. Thus, we can deduce $|B|$ LGTs from C_B, each with different root node. When an incoming node joins the system, it needs to pick up one LGT as guidance to initiate the join process. There are two alternatives: *symmetric*, where every node in the base list can serve as root pivot to guide an incoming node, and *asymmetric*, where only one of them can do so. They are detailed below.

4.2.1. Symmetric join

In the symmetric join process, every node in C_B can serve as the root of an LGT to guide an incoming node. When a node v joins C_B, v chooses an arbitrary node p in B (e.g. the first node it contacts in the system) as the root pivot. This then implicitly determines a unique LGT to be used to guide v's join. Node v finds its neighborhood using the following procedure with p's height as the input parameter i.

Algorithm (S). $v.SearchNeighborhood$(node p, level i)

```
1.    if i = 0
2.      ... insert v beside p ...
3.      return
4.    if p.prev[i] exists
5.      lc := v.Affinity(p.prev[i]))
6.    else lc := ∞
7.    if p.next[i] exists
8.      rc := v.Affinity(p.next[i])
9.    else rc := ∞
10.   mc := v.Affinity(p)
11.   if mc = min{lc, mc, rc}
12.     v.SearchNeighborhood(p, i − 1)
13.   else if rc = min{lc, mc, rc}
14.     v.SearchNeighborhood(p.next[i], i − 1)
15.   else
16.     v.SearchNeighborhood(p.prev[i], i − 1)
```

At each level i, v selects one of the three nodes p, $p.prev[i]$, and $p.next[i]$ to which v is closer, and uses the selected node as new pivot for the next round of comparison. The level is decreased by one at each

round until the lowest level is reached, and then v is inserted as one of the pivot's two level-0 neighbors. In line 2, v is placed in between p and $p.prev[0]$ if the resulting base list has length smaller than placing v in between p and $p.next[0]$ (that is, $d(p.prev[0],v)+d(v,p)+d(p,p.next[0]) \leqslant d(p.prev[0], p)+d(p,v)+d(v,p.next[0])$; see Fig. 8 for an illustration). Note that when F2 is used to calculate the closeness of v to p, an additional parameter i should be provided to $Affinity()$ in lines 5, 8, and 10.

For the complexity, if the height of p is $\log|B|$, then the join process causes $\log|B|$ recursive calls of S. The total number of messages generated is $O(\log|B|)$ if we use F0 and F1 as comparators, and $O(\log^2|B|)$ if F2 is used instead. This is because each distance inquiry $d(x,y)$ requires a message between node x and y, and F0 needs only one inquiry, F1 needs two, while F2 needs $O(\log|B|)$.

In Algorithm S, we do not invoke any role exchange during the neighborhood search process. As commented before, a more greedy approach is to enable this option. To do so, if an incoming node v finds that it is closer to the neighborhood of a pivot p than to the neighborhood of p's two neighbors, and v is closer to the neighborhood of p than p is, then v exchanges the role with p. Node p instead becomes a new node to join the system. The following variation of S implements this join process.

Algorithm (SR). *v.SearchNeighborhoodRE* (node p, level i)

```
1.    ...line 1 to line 10 same as in Algorithm S...
⋮
11.     if mc = min{lc, mc, rc}
12.       if v.Affinity(p) < p.Affinity(p)
13.         ...exchange the role of v and p...
14.         p.SearchNeighborhoodRE(v, i)
15.       else v.SearchNeighborhoodRE(p, i − 1)
16.     else if rc = min{lc, mc, rc}
17.       v.SearchNeighborhoodRE(p.next[i], i − 1)
18.     else
19.       v.SearchNeighborhoodRE(p.prev[i], i − 1)
```

Note that only functions F1 and F2 are meaningful for computing v's closeness to p's neighborhood. Also note that in line 14, p becomes the node to recursively call the neighborhood search process. Its search of new neighborhood will start at the same level. One may also let p start at one level below. Letting p start at level i allows the possibility for p to replace one of its two original neighbors $p.prev[i]$ and $p.next[i]$ when p rejoins, thereby increasing the flexibility of role exchange. The total number of role exchanges incurred by a new node is bounded by the height of the initial root pivot.

4.2.2. Asymmetric join

In the asymmetric join process we use a fixed LGT to guide node joining. For this, we let every newcomer use the same root pivot, say p_0. A newcomer can locate p_0 by invoking a search operation to find a fixed key (say k_0) within p_0's key space range. Once p_0 is located, the join process is the same as the symmetric case. The time and message cost is thus similar to the symmetric case, except that an additional $O(\log|B|)$ message hops are required to locate the root pivot.

However, nodes may come and go in a dynamic environment, and so the root pivot p_0 may not always be available. Since p_0 is located by a fixed key k_0, if p_0 is not available, then the node whose key space range is the closest to k_0 will be used instead. The new root pivot can be located during the search of k_0, as if p_0 is unavailable, then the search mechanism will typically route the search message to the node whose key space range is the closest to k_0. Intuitively, if proximity is respected by the base list, then the use of a pivot with a key space range close to k_0 ensures that the new pivot is also close to the previous one. Therefore, by delegating the closest node to p_0 as the new root pivot, incoming nodes can still have a similar overview of the network topology, thereby allowing nearby nodes to be placed close to each other.

4.3. Selection of overlay construction mechanisms

Recall from Section 3.1 that the structures of SkipNet, skip graph, and Chord can all be viewed as chordal graphs, and thus their construction algorithms can be used to build our overlay. There are, however, subtle differences between them in how they use node IDs. In SkipNet and skip graph, which share the same concept in the overlay design, node IDs are used as membership vectors to determine which rings they belong, thereby to build links between them. More precisely, the overlay is decomposed into a set of rings, each identified by a binary string. A ring of ID x is to host nodes with IDs prefixed by x, and is classified as a level-i ring, where i is the length of x.

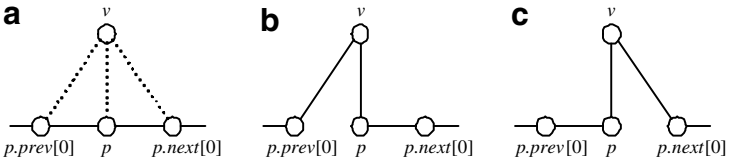

Fig. 8. Join side selection: (a) before v joins; (b) v joins to p's left side; and (c) v joins to the right side.

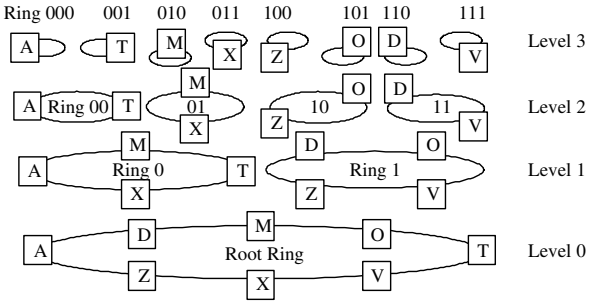

Fig. 9. A perfect SkipNet [18].

For example, Fig. 9 shows a SkipNet from [18]. The ring "0" contains all nodes with an ID of the form 0... The root ring has an empty-string ID, and so hosts all nodes in the system. Node D has ID 110..., and so it belongs to the root ring, ring "1", ring "11", and ring "110". When a new node joins the system, it first chooses a position in the root ring it wishes to sit, say, in between D and M. It then chooses a random binary string as its membership vector to determine the set of rings it belongs, say 0011... The node can use its neighbors in the root ring to help it find its neighbors in ring "0" (i.e., A and M), then the neighbors in ring "00" (A and T), and then the neighbors in ring "001" (T only).[1] A node's neighbors in a level-i ring can be viewed as the node's level-i neighbors in our chordal graphs.

Note again that node IDs are used to help build the net of the rings. There is no correlation between a node's ID and its position in the root ring. Therefore, even if there are a number of nodes all attempting to sit in between nodes D and M in the figure (say, because they are all close to D and M), their joins are likely to cause the rings at higher levels equally populated due to the random binary strings they choose as their membership vectors. Balanced rings also imply balanced load in routing. Thus, introducing proximity-aware joining, as we proposed in the paper, does not cause an unbalanced load in routing.

The Chord overlay also has a chordal graph structure. Like SkipNet, Chord uses node IDs to construct its overlay. Unlike SkipNet, however, Chord requires node IDs to be circularly ordered. If a node x wishes to be inserted in between two nodes u and v, then x must be assigned an identifier $\mathrm{ID}(x)$ that falls in between $\mathrm{ID}(u)$ and $\mathrm{ID}(v)$.[2] Routing tables are constructed so that a level-i link of u points to the node with identifier $\mathrm{ID}(u) + 2^i$. As a result, taking proximity into account in placing nodes might result in an unbalanced load in routing. For example, consider Fig. 10, in which nodes A, B, C, and D are already in the ring, and some other nodes (white and gray) wish to join. Applying proximity-aware node joining may yield a Chord ring with an uneven distribution of nodes, thus an uneven distribution of routing table sizes. Therefore, we will not use the Chord overlay construction mechanism to build a chordal graph.

In addition to the ID-based approach, a chordal graph can be constructed without relying on node IDs (that is, nodes can be "anonymous"!) We observe that in a perfect chordal graph, the successor of v's successor in the base list is v's level-1 successor. That is, $v.next[1] = v.next[0].next[0]$. In general, $v.next[i+1] = v.next[i].next[i]$ and $v.prev[i+1] = v.prev[i].prev[i]$. Therefore, a node's routing table can be constructed starting from the base list and then level by level up, until the node's forward and backward links "cross" each other; that is, at the level

[1] Note that the above join process is slightly different from SkipNet. In SkipNet, a new node builds its routing table starting from the highest level. Skip graph, on the other hand, builds the routing tables bottom up from the base list as described above. Both, however, do not take proximity into account; rather, a new node's position is decided by the object keys it is assigned to handle.

[2] The identifiers also determine the placement of objects in Chord: A node u is to handle all objects whose keys fall in between the identifier (exclusive) of its immediate predecessor in the base ring and $\mathrm{ID}(u)$ (inclusive). In contrast, in the chordal graphs we use, a node's ID has nothing to do with the objects it is responsible for.

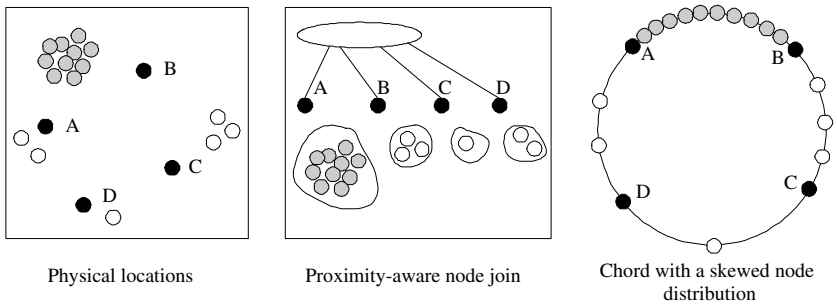

Fig. 10. A skewed structure after applying proximity-aware node joining in Chord.

i where the two key space ranges $[v.upper, v.next[i].upper)$ and $[v.prev[i].lower, v.lower)$ overlap. Section 5.1 will compare the ID-based approach and the anonymous approach in more detail.

5. Simulation

In this section we conduct simulations to evaluate different proximity-aware node join schemes in chordal graphs. Our simulator is written in Java. It uses a single-process, event-driven architecture to simulate concurrent activities of nodes. In the following we first describe the overlay construction and maintenance mechanism, followed by how we model network dynamics. Then, we present the parameters and notations to be measured, simulation setting, and finally the simulation results.

5.1. Overlay construction and maintenance

Constructing and maintaining a chordal graph involve procedures for node joining, leaving, and link maintenance. For node joining, the first node in the system constructs a chordal graph with just the node itself. It has only level-0 links, and the links point to the node itself. Subsequently, a new node uses a join algorithm to find its position in the base list, and "inserts" itself to the list by setting its level-0 links to its successor and predecessor, and informs them to adjust their level-0 links as well.

Recall from Section 4.3 that there are two ways to build a new node's express links. The ID-based approach assigns a random ID to each node, and requires a node v's forward/backward level-i link to point to the first node (clockwise/counterclockwise) in the base list whose ID has the same prefix as ID(v) up to length i. A node v's level-i links can be constructed by traversing along level-$(i-1)$ links. The first node encountered by v that has the same prefix of length i is v's level-i neighbor. The construction of v' routing table ends at the level where v cannot find any other node with the same prefix of longer length. The anonymous approach constructs a node's routing table based on the rule: $v.next[i+1] = v.next[i].next[i]$ and $v.prev[i+1] = v.prev[i].prev[i]$. The construction ends at the level when the node's forward and backward links cross each other.

Note that when a node joins into a chordal graph, some other nodes may have to update their routing tables. In the ID-based approach, the update is easy. This is because all nodes with a common prefix of length i form a doubly-linked list via their level-i links. Thus, when a new node inserts itself into the list, it can also inform its two neighbors to update their level-i links.

For the anonymous approach, however, it is very difficult for a new node to know whose routing tables may need to be updated due to its join. Therefore, we will not immediately fix the express links that are affected by a new node's join. Rather, a background "stabilization" mechanism as in Chord [10] will be invoked periodically to refresh the links. That is, every node periodically rebuilds its express links in the same way as how it constructs them when it joins the system. Note that even if express links are not fixed immediately, as long as the base list is correctly linked, stale express links will not affect search correctness (but may compromise search efficiency), as they are used only to speed up the search process.

Since in the anonymous approach no strong consistency is maintained between express links, concurrent join is easier to implement. In contrast, in the ID-based approach, all nodes with the same prefix of length i are doubly-linked by their level-i links. Maintaining the invariant in the presence of concurrent join is trickier. Moreover, in the ID-based

approach, even if express links can be fixed immediately upon new node join, a stabilization mechanism is still needed to repair them because nodes may fail or leave unexpectedly.

In both approaches, maintaining a connected base list is crucial to the correct functioning of the overlay. To increase fault tolerance, like Chord and Pastry, each node will also maintain a *leaf set* of links that point additionally to a number of nearest successors and predecessors in the base list. Therefore, when a node detects that its successor/predecessor has failed, the nearest successor/predecessor in the leaf set can be used to repair the base list. According to [10], we will maintain about $\log N$ leaf set (half of them forward, and half of them backward). Like the express links, base links (and the leaf set) need to be periodically refreshed by a stabilization mechanism. In general, base links need to be refreshed more frequently than express links, as they are crucial to the correct functioning of the overlay. According to [33], in the simulation we will set the base link stabilization interval to 30 s, and the express link stabilization interval to 3 min.

5.2. Network dynamics

To simulate the dynamics of the system, each node will be given a session time when it connects to the network. As in [33], we use empirical data to model node session time. Saroiu et al. [34] gave a detailed measurement study of the two popular P2P systems Napster and Gnutella. The CDF of session lengths they measured is re-constructed in Fig. 11, and a similar distribution has also been observed in [35]. Here we use T to represent the observation interval, and each session length is represented as a percentage of T. Given the CDF, we can obtain the PDF of session length. Given the PDF, we determine session lengths in our system as follows: when a node joins the network, it selects a session length δ according to the PDF, stays for δ time in the network, and then leaves the network. According to [34], the median session time observed from the Gnutella network is 60 min. This amounts to setting T in Fig. 11 to be 720 min. In our simulation we will also use this value to give a more realistic modeling of node behavior.

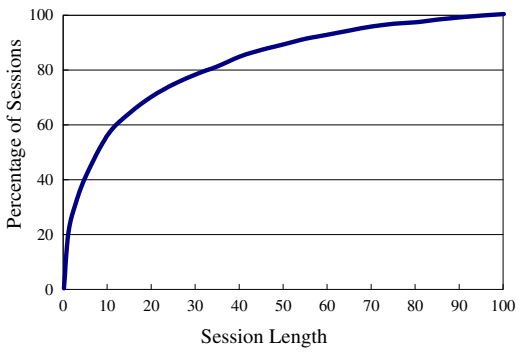

Fig. 11. CDF of system session lengths.

In the simulation, when we need a chordal graph of size N, unless stated otherwise, the overlay is constructed and maintained as follows. Initially, we let N nodes sequentially join the overlay. After the join process is completed, we start to time each node's session, and remove nodes from the overlay when their sessions end. To keep the network size roughly stable, when a node u leaves the overlay, a random node that is currently offline is "awakened" to rejoin the overlay. Similarly, u remains offline until it is awakened by another node. When a node joins the overlay, it randomly contacts an arbitrary node in the overlay as a hookup point to the network. A node leaves the overlay by simply disconnecting all its links to the overlay. Thus, other nodes detect its absence and fix their routing tables only by running the stabilization mechanisms.

5.3. Measurements and notations

Recall the two definitions FD and RT for measuring free drift and routing cost of a chordal graph. For comparison, we define the following metric for measuring the improvement on free drift in a chordal graph C_B constructed with a proximity-aware node join scheme X, over that without the scheme:

$$\mathbf{MD}(\mathrm{X}) = \frac{\mathrm{FD}(C_B^{\mathrm{X}})}{\mathrm{FD}(C_B)} \tag{3}$$

Similarly, the following measures the improvement on routing:

$$\mathbf{MR}(\mathrm{X}) = \frac{\mathrm{RT}(C_B^{\mathrm{X}})}{\mathrm{RT}(C_B)} \tag{4}$$

By the definitions of FD and RT, **MD** concerns the improvement on the average link length in the base list, while **MR** concerns the improvement on the average link length overall.

There are two basic join schemes: symmetric and asymmetric. Role exchange can be further enabled in each scheme, yielding four possible combinations.

We shall use S and A, respectively, to denote the symmetric and asymmetric join schemes without role exchange, and SR and AR to denote the two schemes with role exchange.

In each join scheme, there are three different ways to compute *Affinity*(): F0, F1, and F2. F2 further has two variations: upward-decreasing and uniform. We use the following notations to denote them:

$$\begin{cases} \text{F2(UpDec)} & : \text{F2 with } \Delta = 2^{-i} \\ \text{F2(Uniform)} & : \text{F2 with } \Delta = 1 \end{cases}$$

When combined with the symmetric scheme, we use S–F* to denote a symmetric join scheme with F* for computing *Affinity*(). Thus, S–F2(UpDec) denotes the symmetric join scheme with F2(UpDec). Similarly, A–F2(Uniform) denotes the asymmetric join scheme with F2(Uniform), and AR–F1 denotes the asymmetric join scheme with role exchange and F1 in combination.

In addition, we will also investigate a scheme inspired from Mithos [26]. The scheme is that when a node v joins the system, it selects one random node u from the system as pivot, and compares u's neighbors at the base list in k hops (i.e., the first k nodes $u.next[0]$, $u.next.next[0]$, ... clockwise after u, and the first k nodes $u.prev[0]$, $u.prev.prev[0]$, ... counter-clockwise before u.) If v is closer to u than to any of its neighbors in k hops, then v is inserted beside u (join side selection is as in Fig. 8); otherwise, v uses the closest neighbor as new pivot, and repeats the comparison process. We use "Mithos-k" to denote this scheme. Each round of comparison requires $2k + 1$ probes of distance.

5.4. Simulation setting

We use BRITE [36] to generate several Internet-like physical networks, and then build chordal graphs on top of the physical networks. The link latency between nodes x and y given by BRITE is used to calculate the distance function $d(x, y)$: $d(x, y)$ is the link latency of the shortest path between x and y. BRITE provides several kinds of network topologies. We use the top-down mode in BRITE to first generate the AS-level topology, and then a router-level topology for each node in the AS level (100 nodes per AS-node). For the edge connection mechanism, we use the smallest-degree model, and heavy tailed mode for both inter and intra bandwidth distribution. Node placement is heavy tailed and growth type is incremental. The default network size is 5000, and the default overlay construction mechanism is the random ID approach.

5.5. Simulation results

5.5.1. Comparison of different join schemes

In the first experiment we compare different join schemes by measuring the **MD** and **MR** of the chordal graphs they generate. We consider symmetric and asymmetric joins without role exchange in combination with the four different ways of computing *Affinity*(). In addition, we also investigate Mithos-k for $k = 2, 4, 8, 16$, and 32.

To do the experiment, we generate a physical network of 5000 nodes, and construct a chordal graph on top of the network using a selected node join scheme X. After the overlay construction is completed, the dynamic mechanism discussed in Section 5.2 is activated so that nodes will be on and off alternatively during the simulation interval. In this experiment we set the simulation interval to 2000 min. Recall that the median session time is 60 min. During the simulation interval about half of the nodes will be on and off for more than 30 times. We then snapshot the system at various time spots (where $t = 0$ denotes the initial layout), and measure the average base link length and the average overlay link length at each spot. For a fixed physical network we repeat the experiment at least two times, and calculate the average base link length $\overline{l_B^X}$ and the average overlay link length $\overline{l^X}$ at each observation spot. Similarly, we repeat the experiment to measure the average base link length $\overline{l_B}$ and the average overlay link length $\overline{l}$ at each observation spot without employing any proximity-aware node join scheme; that is, an incoming node simply chooses a random position in the base list to join into the system. The values $\overline{l_B^X}$ and $\overline{l_B}$ are then used to calculate **MD**, while the values $\overline{l^X}$ and $\overline{l}$ are used to calculate **MR**. To ensure that the result is not bound to a specific network topology and is statistically meaningful, we generate five different networks under the same BRITE setting, and repeat the above experiment for each network.[3] We then average the measured **MD** and **MR** at each observation spot. The result is then used as the final **MD** and

[3] Since the average link length is measured over 5000 nodes, the value does not vary much (less than 5%) between different runs of the experiment.

MR at the spot. Fig. 12 shows the resulting **MD** for different join schemes.

From the two charts in Fig. 12, we see that asymmetric join, in general, performs much better than symmetric join. Intuitively, this is because asymmetric join uses a fixed LGT to guide node join. When two nodes join the system, they start from the same root pivot to search their neighborhood. If the two nodes are close to each other, then they will have similar values of $Affinity(p)$ to a pivot p, and so they are likely to follow the same sequence of pivots to reach the same neighborhood. In symmetric join, nodes choose their root pivots at random. When the root pivots are different, the sequence of pivots used during the course of neighborhood search is also different. Hence, even if two nodes are close to each other, the use of different root pivots might cause them to reach different final pivots (although the pivots may still be within some range). The resulting overlay topology thus has higher **MD**.

Of the four different measurements of $Affinity()$, F2(UpDec) and F2(Uniform) both yield a slightly better **MD** than F1 and F0. The difference is more evident with symmetric join. To explain this, we first observe that taking more nodes into account will certainly help for an incoming node to find its neighborhood. Similarly, if possibilities exist for two neighboring nodes in the base list of a chordal graph not to be physically close to each other, then when comparing the closeness of a node v to two different pivots, taking more nodes from the pivot's high level links into account will also help for v to judge which pivot it is closer. Since symmetric join causes more fluctuation in overlay proximity than asymmetric join, the effect of F2(UpDec) and F2(Uniform) in helping improving the overlay proximity is more evident in symmetric join. Between F2(UpDec) and F2(Uniform), F2(Uniform) is slightly better. This implies that different levels of links all contribute to the comparison of a node's closeness to two different pivots, and high level links are no less important than low level ones.

Applying proximity-aware node joining not only reduces the base link length, but also the link length at each level. Fig. 13 shows the average link length at each level for some of the join schemes at $t = 1000$. The result is measured by the ratio of the average link length at each level under a specific join scheme vs. the average link length in the base list under the random join scheme. From the result we see that if proximity is respected by the base list, then the average link length gradually increases as the level increases. In contrast, without a proximity-aware join scheme, the average link length remains approximately the same at each level. Note that because nodes use random IDs to build their

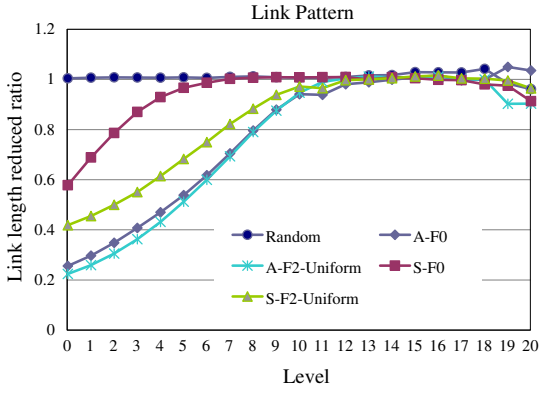

Fig. 13. Link length pattern.

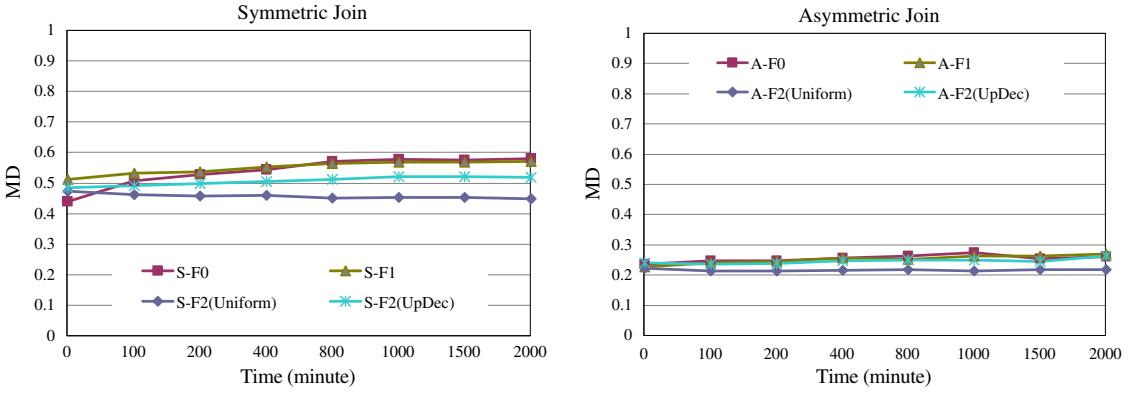

Fig. 12. Measurement of **MD** in symmetric and asymmetric join schemes.

routing tables, each node may have different levels of routing tables. In the experiment, most nodes have at least level-11 links (and recall that a perfect chordal graph of size 5000 should have links up to this level.) After that, the number of nodes having a level-i link decreases significantly. On average, each node can have link level to about 13.6. In Fig. 13 the level is shown up to 20, where less than 1% of nodes still have links at this level. Because the number of nodes having links of this level is significantly fewer, the deviation of the measured value is relatively high. Also note that the link length pattern is similar for other join schemes not shown in the figure, and for the result measured at different time snapshot. The link pattern establishes an important assumption of our hierarchical neighborhood search process, namely, higher level links point to farther nodes. Therefore, an incoming node can use links at different levels to obtain an overview of the network topology.

The corresponding **MR**s for these join schemes are shown in Fig. 14, which shows the overall improvement on the average link length. It is not difficult to see that proximity-aware node joining not only improves the base link length, but also the average link length overall. In general, the better the **MD**, the better the **MR**. We see that a good join scheme can reduce nearly 40% of the average link length.

Fig. 15 shows the results for "Mithos-k". Recall that in the scheme a node v chooses a random node u as pivot, and joins beside u if v cannot find a node within u's k-hop distance that is closer to v than u is to v; otherwise v uses the node that is closer to v as new pivot to repeat the process. Thus, the larger the k, the better the **MD** and **MR**. We note, however, that the overlay proximity tends to become much worse when the dynamic mechanism is activated. For example, for Mithos-8, the initial **MD** is 0.44 (where the overlay is constructed gradually from

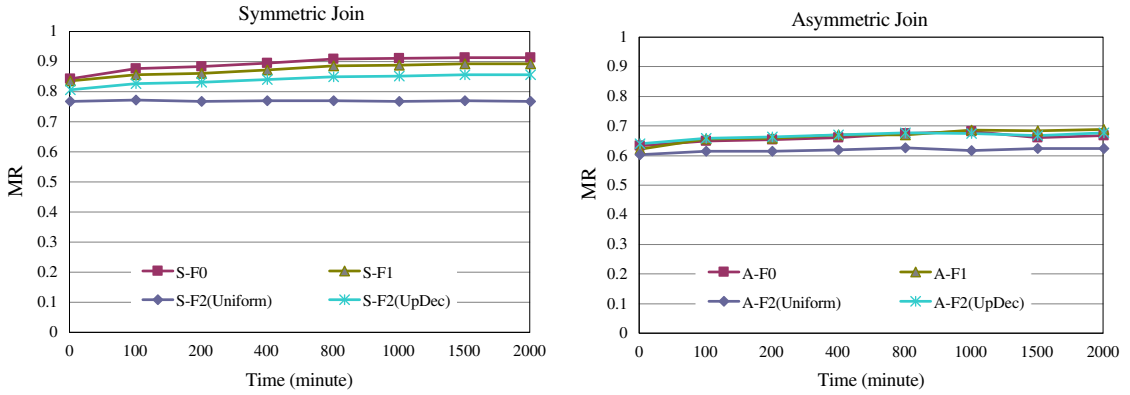

Fig. 14. Measurement of **MR** in symmetric and asymmetric join schemes.

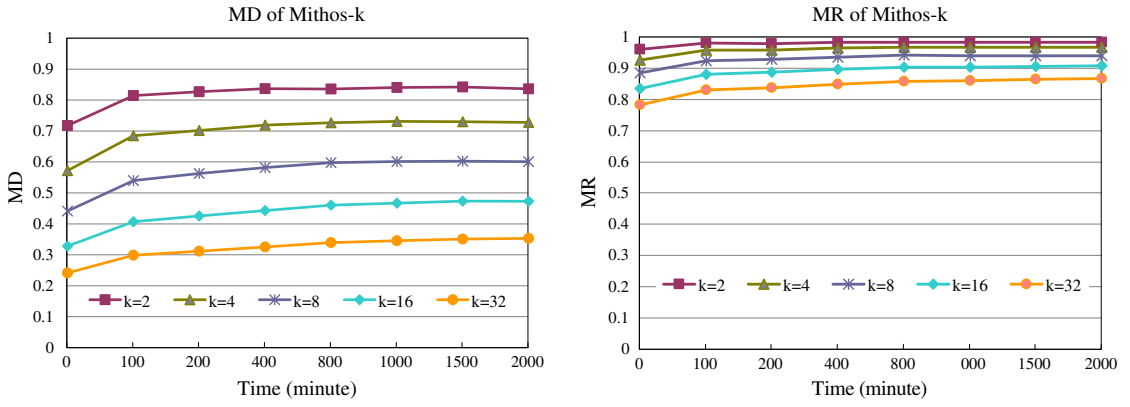

Fig. 15. Measurement of **MD** and **MR** for Mithos-k.

an initially empty network); but it increases to 0.60 when nodes may leave and rejoin the network (where a random pivot will be used to guide a node's rejoin). This can be attributed to the fact that in Mithos-k the search of neighborhood tends to be trapped in an area not far from the initial pivot. For example, consider Fig. 16. If v is to join the system via p as pivot, then when v reaches w, it will find that w's neighboring nodes in two-hop distance is no closer to v than w is to v, and so will stop its neighborhood search and takes a seat beside w. In contrast, our join schemes allow v to search its neighborhood in a wider range through express links. For example, in Fig. 16, when v uses p as pivot, it not only checks its proximity to p, but also its proximity to p's high level neighbors, which are likely to guide v closer to u. Thus, the hierarchy neighborhood search process can yield a better network topology, especially in the asynchronous join scheme.

We also record the distance measuring costs in Mithos-k. Observe that each round of neighborhood search in Mithos-k requires $2k + 1$ inquiries of distance between an incoming node and the pivot's neighboring nodes in k hops. Thus, the larger the k, the higher the costs. The average numbers of distance inquiries per node join for different k's are: 6.76 ($k = 2$), 12.18 ($k = 4$), 22.81 ($k = 8$), 43.76 ($k = 16$), 85.21 ($k = 32$). Note that only one distance inquiry is counted regardless of how many times an incoming node may need to consider its distance with a given node. (A node may be involved in consecutively rounds of neighborhood search.) These data also confirm that the neighborhood search process in Mithos-k tends to be trapped in an area not far from the initial pivot. For example, for $k = 2$, the first round of neighborhood search costs already five distance inquiries, leaving about only one more round to be continued in the search process.

In contrast, our hierarchical neighborhood search process requires about $2h + 1$ distance inquiries if F0 is used for computing $Affinity()$, $4h + 2$ for F1, and about $2h(h + 1) + 2(h + 1)$ for F2, where h is the link level a node can have. In the experiment, a node has an average of link level 13–14. Thus, in terms of distance inquiring costs, F0 and F1 are preferred.

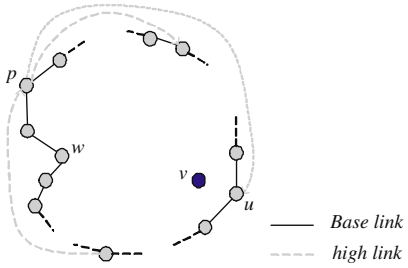

Fig. 16. A node v to join a system.

5.5.2. Impact of role exchange

In the next experiment we study the effect of role exchange, which allows an incoming node v to replace a pivot p if it finds that v can bring better performance than p does at p's position. We choose the following four schemes to illustrate the effect of role exchange: S–F1, S–F2(Uniform), AS–F1, and AS–F2(Uniform). The simulation setting is similar to the previous experiments. The results are shown in Fig. 17. By comparing with the results in Figs. 12 and 14, we see that role exchange does help boost the performance, especially in the symmetric join scheme. For example, S–F1 can yield **MD** to only about 0.56, but with role exchange, the **MD**

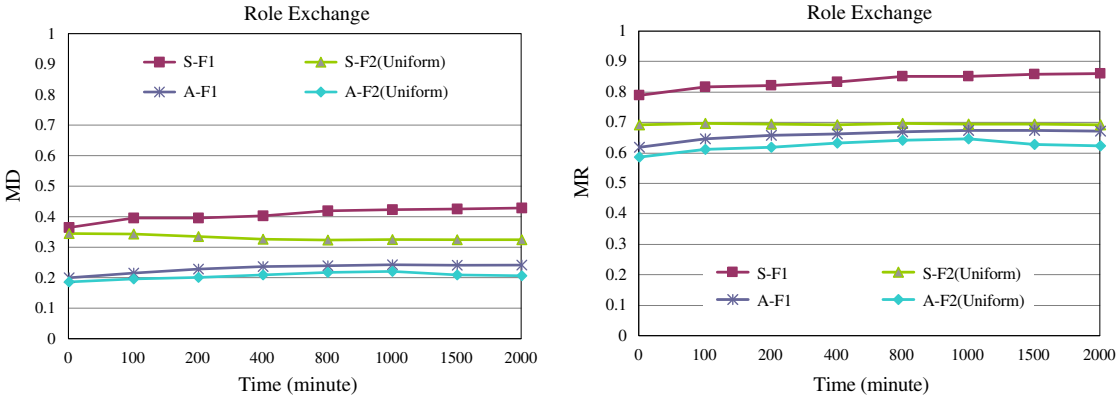

Fig. 17. Effect of role exchanges.

improves to 0.42. For S–F2(Uniform), the **MD** improves from 0.45 to 0.32. For the asymmetric case, role exchange only slightly improves the overlay proximity: AS–F1 from 0.26 to 0.24, and AS–F2(Uniform) from 0.21 to 0.20. This is because the join schemes already result in a very low **MD**, and so further actions can only add a small improvement. We also calculate the number of role exchanges that actually occurred to understand the price to pay. In the 2000-min experiment interval, on average, only 0.89 role exchanges have been performed per node join in S–F1, 1.5 in S–F2(Uniform), 0.48 in AS–F1, and 0.76 in AS–F2(Uniform). Thus, role exchange appears to be an appealing solution to boost the performance of our proximity-aware node join schemes.

5.5.3. Impact of network size

In this experiment we evaluate the impact of network size to different join schemes. We choose the following join schemes to investigate: S–F0, S–F2(Uniform), AS–F0, and AS–F2(Uniform). We vary the network size from 1000 to 10,000, and measure the **MD** and **MR** of the constructed overlay at time $t = 0$. The results are shown in Fig. 18. It turns out that for all the join schemes investigated, they yield the best overlay proximity at the network size 5000 and 7000, and then the performance degrades when the network size becomes smaller and larger. Our explanation for this is that when the network is too crowded or too sparse, it becomes more difficult for a node to estimate and find its neighborhood, and so the resulting MD and MR increase.

5.5.4. Effect of neighbor balance

In this experiment we study the effect of neighbor balance. For the experiment we use the 235,118 vocabulary in the Webster dictionary as our dataset. We construct a chordal graph of 5000 nodes as in Section 5.5.1. Since the selection of a join scheme does not affect the experiment, we simply let a node randomly choose its position to join the network. After the overlay is constructed, we equally partition the key space range (A^*–Z^*) so that each node is responsible for the same size of range. Then we insert the data objects into the overlay. The distribution of object keys is obviously skewed, and so the initial load distribution (number of objects per node) is extremely unbalanced. We then activate the free drift (FD) neighbor balancing mechanism to balance the load. To do so, we randomly select the nodes in sequence to perform free drift. Each selected node u checks if its load versus the load of its level-0 neighbors has exceeded some ratio α. If so, then u shifts some of its objects to the neighbor that has fewer objects to balance their loads. We measure the coefficient of variance (CV) of the load distribution over time.

For the first part of the experiment we assume that the overlay is static (that is, nodes do not leave the network), and we study the effect of free drift by measuring the CV of the load distribution with respect to the total number of neighbor balances performed in the system. The result is shown in Fig. 19, left, for $\alpha = 1.2$ and 1.5. For comparison, we also implement a mechanism that allows a node to arbitrarily choose any level-i neighbor to split their loads, rather than just limited to level-0. Note that this mechanism will destroy object ordering and thus cannot be applied to balance load in chordal graphs. It is used here only to illustrate what we can achieve in load balancing if we have such flexibility. The results of this mechanism are shown in the lines marked as "Arbitrary" in the figure. From

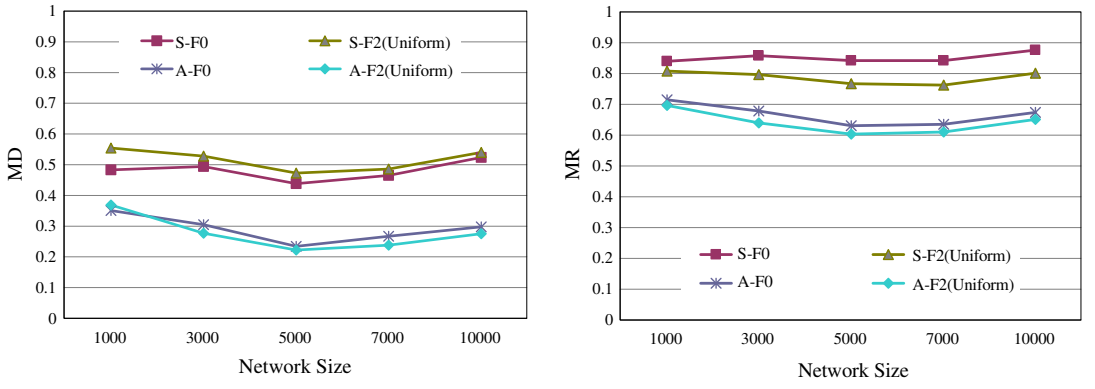

Fig. 18. Impact of network sizes.

the results we see that free drift can effectively reduce the CV from 3.8 to 2.3 after about two neighbor balance operations have been performed per node. After that, however, more neighbor balance operations can only slightly reduce the CV. In contrast, much lower CV can be achieved if a node is able to freely choose any node it is aware of to balance their loads. Thus, maintaining object ordering between neighboring nodes in chordal graphs (as well as in DHT networks) is at the cost of less effective load balancing.

For the second part of the experiment we study free drift in a dynamic environment. As in the first part, we construct an overlay, and then insert the objects to the nodes which have the same size of key space range initially. Then we turn on the dynamic mechanism to allow nodes to leave and rejoin the network. Note that in a dynamic environment some backup mechanism must be installed. Otherwise, when a node leaves the network, its objects will be lost, and, eventually, all objects in the network will be gone. Here we let a node's two level-0 neighbors backup the node's objects so that when the node leaves the network, its objects can still be retrieved from the neighbors. On the other hand, when a node joins the network, it chooses one of the two neighbors that has more objects, and splits the objects with the neighbor to share the load. We let each node periodically invoke the load balancing mechanism at a 3-min interval. Then we measure the CV of the load distribution with respect to time. The result is shown in Fig. 19, right. Again, we see that the CV can be quickly reduced by the free drift neighbor balancing mechanism, but then remains approximately at the same value as time proceeds. Note that load balance is slightly improved in the dynamic environment. This is because node rejoins help to split existing nodes'

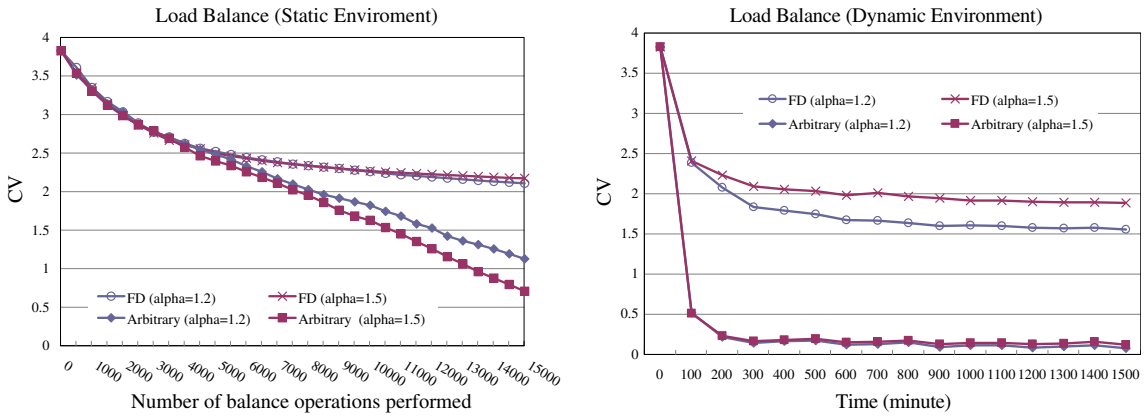

Fig. 19. Effect of load balancing in a static (left) and dynamic (right) environment.

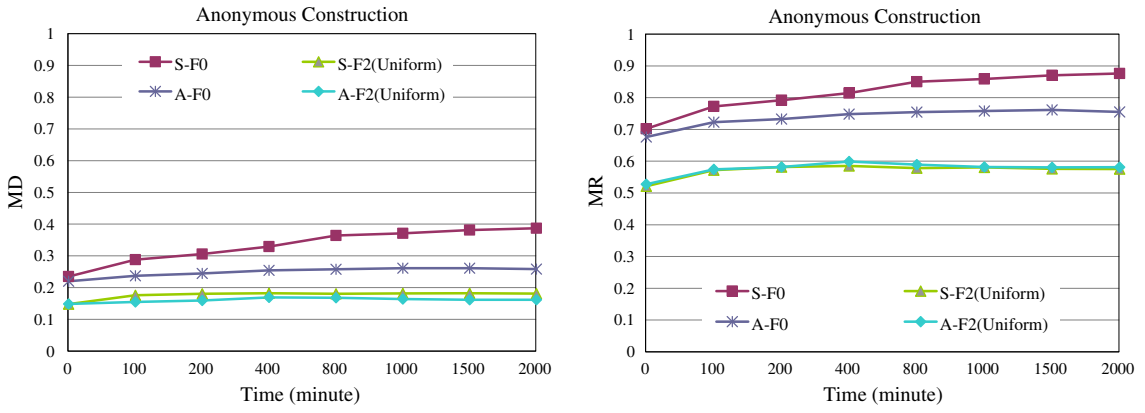

Fig. 20. **MD** and **MR** in anonymous construction.

loads. Also, in the figure a node's load concerns only the number of objects it is responsible for. If we take the backup objects into account, then the resulting CV is still similar, except that the initial CV decreases to 3.

5.5.5. Impact of overlay construction mechanism

In the last experiment we compare different overlay construction mechanisms. So far, our default overlay construction mechanism is to use random IDs to build nodes' routing tables. For comparison, we now use the anonymous approach discussed in Sections 4.3 and 5.1 to construct the overlay. We choose S–F0, S–F2(Uniform), AS–F0, and AS–F2(Uniform) as the node joining schemes. The **MD** and **MR** of the resulting overlays are shown in Fig. 20. By comparing with the results in Figs. 12 and 14, we see that the anonymous approach can yield a lower **MD** and **MR**. For example, for symmetric join, in the random ID approach, S–F0 and S–F2(Uniform) can yield **MD** to only 0.58 and 0.44 at $t = 2000$, while in the anonymous approach, the **MD** improves to 0.38 and 0.18, respectively. For asymmetric join, in the random ID approach AS–F0 and AS–F2(Uniform) can yield **MD** to 0.27 and 0.21, but the anonymous approach improves it to 0.25 and 0.16, respectively.

The reason for the improvement may be attributed to the fact that the anonymous approach tends to build a more balanced link structure. To see this, observe that in the random ID approach, a node v's level-i forward/backward neighbors are the first node clockwise/counterclockwise in the base list that has the same prefix of length i as v. Due to the random assignment of node IDs, although the probability that v's level-i forward and backward links are likely to jump across about the same number (2^i) of nodes in the base list, the possibility exists that they jump across different number of nodes. To illustrate, the ID assignment in Fig. 21 will yield the node 0110... to have an unbalanced link structure shown in Fig. 21a, as opposed to what a perfect chordal graph in Fig. 21b should be. Clearly, the unbalanced link structure will affect the precision of an incoming node's estimate of its neighborhood in the overlay. In contrast, the anonymous approach builds a node's routing table level by level up, with each level jumps about twice of the level below. Therefore, a more balanced link structure can be constructed.

An unbalanced link structure also affects average overlay hop count in search. To illustrate this, we measure the average number of overlay hops need to be traversed from a node v to a node responsible for a target key. We assume that the querying node v is arbitrary and keys are uniformly distributed. Note that when searching for a key, there are two alternatives: "One-Way", where queries can be forwarded in only one direction (say, clockwise) in the base list as in Chord; and "Two-Way", where both forward and backward links can be used to approach to the target. Obviously, Two-Way search requires less hop count. We use S–F0 as the node join scheme (although the results should not depend on the join scheme), and measure the average overlay search hop count in both the static environment (where nodes do not leave the network) and the dynamic environment (where nodes may leave and rejoin the network after the initial overlay has been built). The results are given in Table 1. Each hop

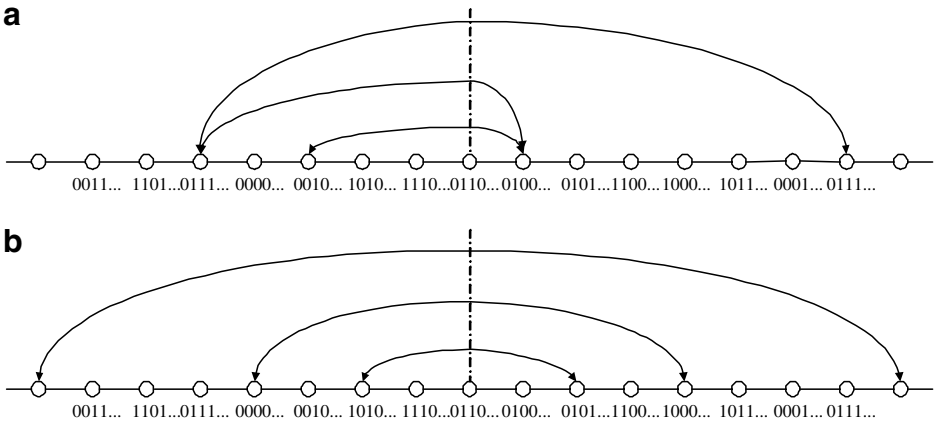

Fig. 21. (a) An assignment of node IDs that causes an unbalanced link structure at node 0110...; (b) a perfect chordal graph.

Table 1
Average hop count

Search	Static		Dynamic	
	Anonymous	Random ID	Anonymous	Random ID
One-Way	5.97	10.98	7.84	11.20
Two-Way	5.46	10.14	6.45	10.14

count is an average of 20,000 queries issued to the network.

From the results we see that the anonymous approach can yield a much smaller hop count than the random ID approach in both the static and dynamic environments. Note that in a static environment, the anonymous approach is able to construct a perfect choral graph. The expected one-way search hop count should be about $1/2\log 5000 \approx 6.14$, which is close to our measurement 5.97. When the network becomes dynamic, the perfect structure of the choral graph can no longer be guaranteed. A long hop jump in the search process may be replaced by a number of small-hop jumps, thus increasing the search hop count.

In addition to the search hop count, we also measure the average link levels a node can have, which in turn affects the join cost. On average, the random ID approach results in a routing table of links up to level 13.6 in both the static and dynamic environments; while the anonymous approach yields the link level to 11 in the static environment, and 12 in the dynamic case. Thus, the anonymous approach also helps to reduce the join cost.

6. Conclusions and future work

We have presented a platform, called *chordal graph*, for building a range-queriable system. Like DHTs, search is guaranteed with reasonable costs. Unlike DHTs, however, objects are not bound to any particular node. Rather, they can reside at any node as long as some ordering between objects in different nodes is respected. This allows objects to be freely moved between nodes to balance load. Decoupling the link between nodes and objects allows the overlay network to be constructed independently of object placement, focusing only on its own concerns, e.g. proximity and balanced load in routing.

The chordal graph structure is not new. In fact, some existing overlay construction mechanisms like SkipNet and skip graph are able to construct a chordal graph. Our main contribution is to present several heuristics for proximity-aware node joining to reduce communication costs in the platform. These heuristics are based on a novel process called *hierarchical neighborhood search*, which utilizes the overlay links to provide incoming nodes an overview of the network topology, thereby to guide them to find their neighborhood in the overlay. Variations of the join scheme arise when different methods are used to estimate the closeness of a node to an area, whether or not the same sequence of pivots are used to guide newcomers, and whether or not existing nodes' positions can be exchanged.

We have conducted a comprehensive analysis of the heuristics via simulation. The results indicate that asymmetric join in general yields much better overlay proximity than symmetric join. Of the different closeness measurements, F2(Uniform) outperforms the others. However, in many cases the gap is not very significant. Considering the join cost (F2 requires $O(\log^2 N)$ distance lookups while F0 and F1 require only $O(\log N)$, where N is the network size), F0 and F1 may still be attractive. Moreover, role exchange can further boost the performance, especially for symmetric join schemes that are not able to generate an overlay proximity as good as asymmetric ones. The cost of role exchange is not high: on average no more than two exchanges per join. Overall, a good join scheme can reduce as much as 80% of the communication costs between neighboring nodes, and 40% overall.

As we said before, existing overlay construction mechanisms like SkipNet and skip graph are able to construct a chordal graph. SkipNet and skip graph use random IDs to help building a node's routing table. We have also presented an anonymous approach that does not rely on node IDs to construct the overlay. Compared with the random ID approach, the anonymous approach does not impose a constraint like a node's level-i neighbors are the closest nodes in the base list with the same ID prefix of length i. This allows neighbor tables to be easier to maintain. Moreover, the anonymous approach is able to generate a more balanced link structure like a perfect chordal graph. As a result, the anonymous approach can yield not just a better overlay proximity, but can also reduce the join costs and the average overlay search hop count.

For future work, we recall that objects in a chordal graph are not bound to a particular node, but can be freely moved between neighboring nodes to balance load. In the simulation, we showed that this

neighbor balancing scheme is effective. However, when compared to a balancing scheme that allows a node to arbitrarily choose another node to balance their loads, we see that the free drift neighbor balancing scheme has its limitation. Although this "arbitrary" balancing scheme cannot be directly applied in choral graphs (as well as in DHTs), it does indicate that there is still some room to improve for load balancing in chordal graphs.

Finally, we have treated fault tolerance as a separate issue from the paper. A good replication scheme is important for proximity-aware range query. This is because when sequential objects are placed in nearby nodes, a regional network collapse may block queries of some interval. So simply replicating each node's data to its neighbors does not enhance reliability much. Rather, replication is better done through express links to place an object's replica far away to survive local catastrophe. Future work will also investigate this issue in detail.

Acknowledgements

The author would like to thank Chih-Ho Kang for initial discussion of the paper and for drawing some of the figures in the paper, and Man-Ling Hung and Wing-Tat Wong for conducting the simulation. He would also like to thank the anonymous referees for their valuable comments and suggestions.

References

[1] I. Clarke, O. Sandberg, B. Wiley, T.W. Hong, Freenet: A distributed anonymous information storage and retrieval system, in: Proceedings of the 2000 International Workshop on Design Issues in Anonymity and Unobservability (DIAU 2000), Lecture Notes in Computer Science, 2009, Springer-Verlag, 2001, pp. 46–66.

[2] M. Ripeanu, Peer-to-peer architecture case study: Gnutella network, in: Proceedings of the First International Conference on Peer-to-Peer Computing (P2P 2001), IEEE Computer Society, 2001, pp. 99–100.

[3] B. Yang, H. Garcia-Molina, Improving search in peer-to-peer systems, in: Proceedings of the Twenty-Second International Conference on Distributed Computing Systems (ICDCS 2002), IEEE Computer Society, 2002, pp. 5–14.

[4] A. Crespo, H. Garcia-Molina, Routing indices for peer-to-peer systems, in: Proceedings of the Twenty-Second International Conference on Distributed Computing Systems (ICDCS 2002), IEEE Computer Society, 2002, pp. 23–32.

[5] Q. Lv, P. Cao, E. Cohen, K. Li, S. Shenker, Search and replication in unstructured peer-to-peer networks, in: Proceedings of the 16th International Conference on Supercomputing (ICS), ACM Press, 2002, pp. 84–95.

[6] H. Cai, J. Wang, Foreseer: a novel, locality-aware peer-to-peer system architecture for keyword searches, in: Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware, Lecture Notes in Computer Science, vol. 3231, Springer-Verlag, 2004, pp. 38–58.

[7] A.-H. Cheng, Y.-J. Joung, Probabilistic file indexing and searching in unstructured peer-to-peer networks, Computer Networks 50 (1) (2006) 106–127.

[8] B.Y. Zhao, J. Kubiatowicz, A.D. Joseph, Tapestry: an infrastructure for fault-tolerant wide-area location and routing, Tech. Rep. UCB/CSD-01-1141, University of California, Berkeley, April 2001.

[9] A. Rowstron, P. Druschel, Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems, in: Proceedings of the 2001 IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001), Lecture Notes in Computer Science, vol. 2218, Springer-Verlag, 2001, pp. 329–350.

[10] I. Stoica, R. Morris, D.R. Karger, M.F. Kaashoek, H. Balakrishnan, Chord: A scalable peer-to-peer lookup service for Internet applications, in: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM 2001), ACM Press, 2001, pp. 149–160.

[11] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker, A scalable content-addressable network, in: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM 2001), ACM Press, 2001, pp. 161–172.

[12] B. Awerbuch, C. Scheideler, Peer-to-peer systems for prefix search, in: Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing (PODC 2003), ACM Press, 2003, pp. 123–132.

[13] K. Aberer, M. Hauswirth, M. Punceva, R. Schmidt, Improving data access in P2P systems, IEEE Internet Computing 6 (1) (2002) 58–67.

[14] M. Cai, M. Frank, J. Chen, P. Szekely, MAAN: A multi-attribute addressable network for grid information services, in: Proceedings of the Fourth International Workshop on Grid Computing (GRID'03), IEEE Computer Society, 2003, pp. 184–191.

[15] A. Andrzejak, Z. Xu, Scalable, efficient range queries for grid information services, in: Proceedings of Second International Conference on Peer-to-Peer Computing (P2P 2002), IEEE Computer Society, 2002, pp. 33–40.

[16] S. Ramabhadran, S. Ratnasamy, J.M. Hellerstein, S. Shenker, Brief announcement: prefix hash tree, in: Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC'04), ACM Press, 2004, p. 368.

[17] A. Datta, M. Hauswirth, R. John, R. Schmidt, K. Aberer, Range queries in trie-structured overlays, in: Proceedings of The Fifth IEEE International Conference on Peer-to-Peer Computing (P2P 2005), IEEE Computer Society, 2005, pp. 57–66.

[18] N.J.A. Harvey, M.B. Jones, S. Saroiu, M. Theimer, A. Wolman, SkipNet: A scalable overlay network with practical locality properties, in: Proceedings of the Fourth USENIX Symposium on Internet Technologies and Systems (USITS 2003), 2003, pp. 113–126.

[19] J. Aspnes, G. Shah, Skip graphs, in: Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms

(SODA 2003), Society for Industrial and Applied Mathematics, 2003, pp. 384–393.
[20] J. Aspnes, J. Kirsch, A. Krishnamurthy, Load balancing and locality in range-queriable data structures, in: Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC 2004), ACM Press, 2004, pp. 115–124.
[21] D.R. Karger, M. Ruhl, Simple efficient load balancing algorithms for peer-to-peer systems, in: Proceedings of the 16th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 2004), ACM Press, 2004, pp. 36–43.
[22] P. Ganesan, M. Bawa, H. Garcia-Molina, Online balancing of range-partitioned data with applications to peer-to-peer systems, in: Proceedings of the 30th International Conference on Very Large Data Bases (VLDB 2004), Morgan Kaufman, Toronto, Canada, 2004, pp. 444–455.
[23] A.R. Bharambe, M. Agrawal, S. Seshan, Mercury: supporting scalable multi-attribute range queries, in: Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM 2004), ACM Press, 2004, pp. 353–366.
[24] S. Ratnasamy, M. Handley, R. Karp, S. Shenker, Topologically-aware overlay construction and server selection, in: Proceedings of 21st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2002), vol. 3, 2002, pp. 1190–1199.
[25] M. Waldvogel, R. Rinaldi, Efficient topology-aware overlay network, in: Proceedings of HotNets-I, 2002.
[26] M. Waldvogel, R. Rinaldi, Efficient topology-aware overlay network, SIGCOMM Computer Communication Review 33 (1) (2003) 101–106.
[27] X.Y. Zhang, Q. Zhang, Z. Zhang, G. Song, W. Zhu, A construction of locality-aware overlay network: overlay and its performance, IEEE Journal on Selected Areas in Communications 22 (1) (2004) 18–28.
[28] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, I. Stoica, The impact of DHT routing geometry on resilience and proximity, in: Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM 2003), ACM Press, 2003, pp. 381–394.
[29] M. Castro, P. Druschel, Y.C. Hu, A. Rowstron, Proximity neighbor selection in tree-based structured peer-to-peer overlays, Tech. Rep. MSR/TR-2003-52, Microsoft Research, June 2003.
[30] P. Ganesan, Q. Sun, H. Garcia-Molina, Apocrypha: making P2P overlays network-aware, Tech. Rep., Stanford University, November 2003.
[31] H. Zhang, A. Goel, R. Govindan, Incrementally improving lookup latency in distributed hash table systems, in: Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2003), ACM Press, 2003, pp. 114–125.
[32] N.J.A. Harvey, J.I. Munro, Deterministic SkipNet, in: Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC), 2003, pp. 152–153.
[33] Y.-J. Joung, J.-C. Wang, Chord^2: a two-layer chord for reducing maintenance overhead via heterogeneity, Computer Networks 51 (3) (2007) 712–731.
[34] S. Saroiu, P.K. Gummadi, S.D. Gribble, A measurement study of peer-to-peer file sharing systems, in: Proceedings of the 2002 Multimedia Computing and Networking (MMCN 2002), The International Society of Optical Engineering, 2002.
[35] K. Chu, K. Labonte, B.N. Levine, Availability and locality measurements of peer-to-peer file systems, in: Proceedings of SPIE (SPIE 2002), vol. 4868, The International Society for Optical Engineering, 2002.
[36] A. Medina, A. Lakhina, I. Matta, J. Byers, BRITE: Universal topology generation from a user's perspective, Tech. Rep. BUCS-TR-2001-003, Boston University, April 2001.

Yuh-Jzer Joung received his B.S. in Electrical Engineering from the National Taiwan University in 1984, and his M.S. and Ph.D., in Computer Science from the State University of New York at Stony Brook in 1988 and 1992, respectively. He is currently a professor at the Department of Information Management in the National Taiwan University, where he has been a faculty member since 1992. From 1999 to 2000, he was a visiting scientist at the Lab for Computer Science, Massachusetts Institute of Technology. He was the chair of his department from 2001 to 2005. His main research interests are in the area of distributed computing, with specific interests in multiparty interaction, fairness, (group) mutual exclusion, ad hoc and peer-to-peer computing.