Available online at www.sciencedirect.com
ScienceDirect
Computer Networks 51 (2007) 712–731
Computer Networks
www.elsevier.com/locate/comnet

Chord^2: A two-layer Chord for reducing maintenance overhead via heterogeneity ☆

Yuh-Jzer Joung *, Jiaw-Chang Wang

Department of Information Management, National Taiwan University, Taipei, Taiwan, ROC

Received 29 August 2005; received in revised form 18 March 2006; accepted 31 May 2006
Available online 5 July 2006

Responsible Editor: R. LoCigno

Abstract

Empirical studies have shown that participating nodes in peer-to-peer (P2P) systems are not equivalent. Some nodes, known as "super peers", are more powerful and stable than the others. Such heterogeneity has been taken into account in the design of P2P systems in two ways: by employing super peers to serve as index servers for query, and by routing through super peers to speed up query. In this paper, we use super peers to reduce maintenance cost in Chord—a DHT network which, like other DHT-based systems, is often praised for its guaranteed search feature but has relatively higher maintenance overhead than Gnutella-like unstructured P2P networks.

Keywords: Peer-to-peer (P2P); Distributed hash table (DHT); Chord; Structured overlay; Heterogeneity; Overlay maintenance

1. Introduction

Most structured peer-to-peer overlay networks operate as *distributed hash tables* (*DHTs*). The basic idea is to associate objects with unique identifiers as their keys. Each participating node also has a unique identifier. There is a globally known function that maps object keys to node identifiers so that objects are inserted into the network by placing them (or their indexes) in the nodes responsible for their keys. So, locating an object becomes a routing problem from the requesting node to a destination identifier. To route messages, each node maintains some information about other nodes as guide to forward queries. There have been many ingenious designs on DHTs to provide efficient queries that operate in $O(\log N)$ overlay hops by maintaining only $O(\log N)$ routing table entries, e.g., Chord [1], CAN [2], Kademlia [3], Tapestry [4], Pastry [5], and Viceroy [6].

Although DHT based P2P systems are praised for their efficiency in search, most existing P2P systems are still based on some simple design. For example,

☆ Research supported in part by the National Science Council, Taipei, Taiwan, Grants NSC 93-2213-E-002-096 and NSC 94-2213-E-002-036. A preliminary version of this paper appeared in *Proc. 5th International Workshop on Global and Peer-to-Peer Computing*, Cardiff, UK, May 2005.
* Corresponding author. Tel.: +886 2 33661183; fax: +886 2 23627094.
E-mail address: joung@ntu.edu.tw (Y.-J. Joung).

1389-1286/$ - see front matter
doi:10.1016/j.comnet.2006.05.010

Napster, eMule[1], and KaZaA[2] use one or more centralized servers for indexing objects, while Gnutella[3] and Freenet [7] allow peers to be loosely coupled in a fully distributed but unstructured manner. P2P systems with some centralized index servers are hard to scale and exhibit single point of failure. Unstructured P2P systems are named because they do not deliberately maintain their network in a specific structure other than to keep it connected. As a result, there is much freedom for nodes to choose other nodes as entries in their routing tables. So node joining and leaving pose little problem to the network, and so the systems are highly robust and easily maintained. However, the search costs could be very high as search in an unstructured network is more or less an exhaustive process, and in the worst case the entire network must be traversed. That is, search cannot be guaranteed in bounded costs.

DHTs, on the other hand, are very sensitive to the dynamics of the network [8]. This is because routing efficiency in DHTs is based on the consistent and cooperative maintenance of routing tables at each node to keep the network structure in shape. Most DHTs do not have an efficient way of maintaining their networks. For example, in Chord, $O(\log^2 N)$ messages are needed periodically for a node to maintain its routing table (see Section 3.2). The higher the dynamics of the network, the shorter the interval to refresh routing tables, and so the higher the maintenance costs.

The original DHT designs tend to treat nodes equally, and thus emphasize load balancing among them. However, empirical studies have shown that diversity exists among participating nodes in P2P systems. For example, Saroiu et al. [9] observed that 50% of Gnutella nodes have session time less than 60 min, and that 10% of nodes are relatively more stable than the others. Indeed, existing P2P systems like Morpheus, KaZaA, and eMule have all taken heterogeneity into account and employed "super peers" that are more powerful than ordinary peers to act as regional centralized index servers. Other systems, e.g., Brocade [10] and Expressway [11], use network access points as super peers to help routing as they are often very stable and with high bandwidth.

In this paper, we take a third approach for super peers: employ them to reduce maintenance costs in DHTs. We use Chord as an example. In Chord, the main overhead in maintenance is due to the periodical refreshing of routing tables, which takes $O(\log^2 N)$ messages per node per refreshing. The refreshing frequency is determined by the stability of the network. If nodes are relatively stable, refreshing can be done less often, thus causing less overhead; otherwise, refreshing needs to be performed more frequently, thereby yielding higher costs. Rather than letting nodes periodically refresh their routing tables, we propose a two-layer structure called **Chord^2**. The lower layer is the regular Chord ring, while the upper layer is a ring for maintenance. Like Chord, we let nodes in the regular ring periodically probe their successors, which costs only $O(1)$ messages per activation. When (and only when) a change is detected, some super peer will be informed to notify affected nodes ($O(\log N)$ on average) to update their routing tables, thus reducing the maintenance costs of the regular ring to $O(\log N)$. The maintenance layer is also implemented as a Chord ring. However, since it is constructed from super peers that are more stable than ordinary nodes, the maintenance costs of the ring are relatively low compared to an equivalent ring with diverse characteristics.

The rest of the paper is organized as follows. Section 2 discusses related work; Section 3 gives a brief introduction to Chord; Section 4 presents Chord^2; and Section 5 presents the experiment results. Conclusions are offered in Section 6.

2. Related work

There have been several approaches to cope with heterogeneity in P2P systems. The most popular way is to cluster peers, and select a super peer in each cluster as a local server to manage the cluster as well as to index objects in the cluster. Intra-cluster communication and lookup within a cluster can therefore be efficiently performed via the super peer. The super peers form an overlay to facilitate inter-cluster communication. The overlay is typically unstructured. Examples include KaZaA and recent version of Gnutella [12]. Some guidelines for designing this kind of super peer networks are discussed in [13], including the ratio of clients to super peers, super peer redundancy, and out-degree of super peers.

In the above approach only one overlay has been constructed; ordinary peers themselves do not form another overlay. In contrast, a secondary overlay has also been proposed over an existing overlay to assist routing. For example, Brocade [10] organizes

[1] http://www.emule-project.net/.
[2] http://www.kazaa.com.
[3] http://www.gnutella.com/.

a structured overlay over another structured overlay to speed up routing, as the super peers are chosen from some network access points such as gateways and routers. Expressway [11] builds the auxiliary overlay by further taking proximity into account. HONet [14] also uses a similar approach, but the lower level overlay is composed of clusters, each with its own structure. Cluster heads form the secondary overlay, also structured. In addition, random connections between members across clusters are added as shortcuts to further improve routing. An extreme approach is taken in [15] to push all the lookup load into a small number of super peers and to have each maintain a global index table so as to achieve constant-time lookup.

Rather than adding just one extra overlay, hierarchical overlays have also been explored [16,17]. In these systems, peers are first clustered, each with its own structure like the HONet discussed above. Within each cluster one or more peers are selected as members of the second layer. Similarly, the second-layer members are also clustered, and some of them are selected to form the third layer, and so on. Advantages of using hierarchy include scalability, administrative autonomy, adaption to the underlying physical network, and hierarchical access control. The work of [16] allows clusters to have different structure so that existing DHTs can be linked together without modifying their structure. Heterogeneity is also taken into account when selecting super peers for higher level clusters. Super peers are used mainly for inter-cluster routing. Canon [17], on the other hand, assumes homogeneous structure for all clusters, and designs some ingenious algorithm to merge clusters (in particular, Chord rings) into a larger one. Note that the members of a leaf cluster are also the members of every ancestor cluster. So node heterogeneity is not considered in Canon. Both [16,17] concern DHTs. Hierarchical unstructured overlays are considered in [18,19].

In contrast to the above systems that take heterogeneity into account to help object indexing and routing, we utilize heterogeneity to build a maintenance infrastructure in DHTs to reduce the costs. Several proposals for reducing maintenance costs in DHTs have also appeared in the literature, although they do not approach the problem from peer heterogeneity as we do. In [20], every node records its average session length for some most recent appearances. When a node wishes to update its routing table entries, the preference is given to the peer with longer session length if there is more than one candidate to fill an entry. In [21], a self-tuning strategy for Pastry is proposed to achieve the desired routing reliability while minimizing the maintenance costs. To achieve this goal, they dynamically modulate the time interval for each node sending liveness probes to each entry in its routing table according to the estimation of the node failure rate and network size. EpiChord [22] amortizes Chord maintenance costs into existing lookups by piggybacking network information on query replies to keep nodes' routing states up-to-date. In addition, each node uses virtually unlimited storage to cache the global identifier space it has learned, and issues parallel queries to reduce lookup latency as well as to avoid costly lookup timeouts. Under lookup-intensive workloads, EpiChord is able to achieve O(1)-hop lookup, and at least $O(\log N)$-hop lookup performance under churn-intensive workloads, where N is the network size. In short, EpiChord relies on high lookup traffic to support the desired level of performance.

3. Chord

Chord [1] is a distributed lookup protocol that provides the following basic operation in a P2P network: maps a given key onto a node. Data location can then be implemented by assigning a key with each data item, and storing the key at the node which the key maps to. Node identifiers are obtained by hashing their IPs and ports into m-bit integers ordered in a ring modulo 2^m. A key k is assigned to the first node whose identifier is equal to or follows it (clockwise) in the ring. This node is called the *successor* node of key k, denoted by *successor*(k).

To ensure that queries of object keys can be resolved in the ring, every node maintains the link to its successor in the ring. So the successor node of key k can be determined by traversing through the successor links until the node *successor*(k) is reached. To speed up the search process, each node also maintains $m-1$ *fingers*, where the ith finger points to the node 2^i away in the identifier ring, $0 < i \leqslant m-1$. (Successor can be viewed as the zeroth finger.) See Fig. 1 for illustration.

Since every node has a finger pointing at least halfway to any destination key, each query can be returned with a sequence of halfway forwarding in at most $\log N$ steps, where N denotes the network size. When nodes join, leave, or fail, some fingers

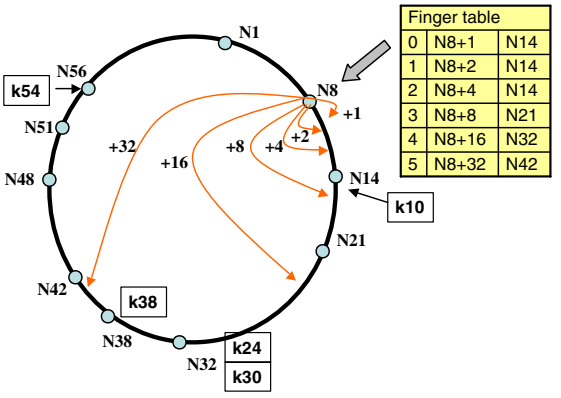

Fig. 1. A Chord finger table.

must be refreshed to guarantee the $\log N$ bound in routing. To deal with this, Chord introduces an overlay maintenance protocol, which is detailed in the following section.

3.1. Chord maintenance algorithm

A node n joins the Chord network by contacting any existing node n', and asks n' to find n's successor in the ring. Then n is integrated into the Chord ring by setting it successor link to point to the successor, and then builds its finger table with the help of the successor.

```
// node n joins through node n'
n.join (n')
    predecessor := nil;
    s := n'.find_successor(n);
    successor := s;
    build_fingers(s);

// ask node n to find the successor of x
n.find_successor(x)
    if (x ∈ (n, n.successor])
        return n.successor;
    else
        n' := closest_preceding_node(x);
        return n'.find_successor(x);

// search finger table for the highest predecessor
// of x
n.closest_preceding_node(x)
    for i := m − 1 downto 1
        if (finger[i] ∈ (n, x))
            return finger[i];
    return n;
```

```
// build finger table
n.build_fingers(s)
    i0 := ⌊log(successor − n)⌋ + 1;
    for i0 ⩽ i < m − 1
        finger[i] := s.find_successor(n + 2^i);
```

After n joins the system, some nodes have to update their fingers originally pointing to n's successor now to n. Because fingers are not symmetric, nodes have no idea when fingers should be refreshed. In Chord, this is solved by letting each node periodically execute *fix_fingers* to keep fingers up to date.

```
// periodically refresh finger table entries
n.fix_fingers()
    build_fingers(n);
```

Note that in the above joining process, after n has located its successor s, only the successor link of n (to s) has been set. The predecessor links of n and s as well as the successor link of n's predecessor are set by letting nodes periodically execute the *stabilization* process. This process also allows existing nodes to learn new nodes that have concurrently joined the system.

```
// periodically probe n's successor s and inform s
// of n
n.stabilization()
    check_predecessor();
    x := successor.predecessor;
    if (x ∈ (n, successor)) // successor has changed
                            // due to new joining
        successor := x;
    successor.notify(n);

// n notifies s to be s's predecessor
s.notify(n)
    if (predecessor = nil or n ∈ (predecessor, s))
        predecessor := n;
```

When a node leaves or fails, its departure must be detected to maintain the connection of the ring. This is done by letting processes periodically check their predecessors and successors. In addition, to enhance the robustness of the system, rather than keeping a single successor node, each node keeps a successor list of the first $\log N$ nodes following it in the ring. Each node maintains its successor list by periodically fetching the successor list of its successor s, appending s to the first of the list, and removing the last entry.

```
// periodically check whether predecessor has
// failed
n.check_predecessor()
    if (predecessor has failed)
        predecessor := nil;

// periodically reconcile with successor's succes-
sor list
n.fix_successor_list()
    ⟨s_1, ..., s_r⟩ := successor.successor_list;
    successor_list := ⟨successor, s_1, ..., s_{r-1}⟩;

//periodically update failed successor
n.fix_successor()
    if (successor has failed)
        successor := smallest alive node in successor_
        list;
```

3.2. Analysis

The correctness of lookup relies on the correctness of the successor links, while the efficiency of lookup relies on the correctness of the finger tables. This is because when a node forwards a lookup message to its ith finger node but the finger node has failed, the lookup message will be re-transmitted to its $(i-1)$th finger node. So instead of making a long jump in the ring during a lookup process, more steps of short jumps would be required. So stale fingers increase the costs of lookup which would otherwise be in $\log N$ steps. The shortest step a node can jump is by its successor link. So if successor links are incorrect, lookup cannot be guaranteed. In the Chord maintenance algorithm, the process of *stabilization*() is used to maintain successor links, while the process of *fix_fingers*() is to maintain finger tables. An execution of *stabilization*() costs four messages, while an execution of *fix_fingers*() costs $O(\log^2 N)$ messages (because $\log N$ executions of *find_successor*() are generated, each of which costs at most $\log N$ messages). So the maintenance of finger tables accounts for most of the overhead.

4. Chord^2

In the Chord maintenance protocol, each node must periodically verify its successor (and predecessor) links and fingers. As noted in the previous section, the maintenance of successor links is crucial to the correctness of lookup function, while the maintenance of fingers affects only lookup efficiency. Also, the latter accounts for most of the maintenance overhead.

To reduce the maintenance costs, we suggest imposing a **conduct layer** over the regular Chord ring (see Fig. 2, top). The maintenance of fingers is removed from the Chord ring, allowing only the periodical check of successor/predecessor links in the ring. If a change is detected (meaning that some node has joined/left the system), this change will be reported to the conduct layer, which is then responsible for notifying affected nodes to update their fingers.

To implement the conduct layer also in a peer-to-peer style, the layer can be constructed from nodes that are more powerful and stable. This is feasible as empirical studies have shown that nodes in P2P networks are indeed heterogeneous. The advantage of using more stable nodes to construct the conduct layer is that the cost to maintain the layer is lower than the cost to maintain a network of the same structure but with diverse characteristics. For example, if the conduct layer is also implemented as a Chord ring, then the use of more stable nodes can reduce the maintenance costs of the conduct ring by increasing the interval of the periodical check of successor links and fingers.

In our design, the maintenance layer is indeed implemented also as a Chord ring, and so we call the new system **Chord^2**. See Fig. 2, left. For a distinguishing purpose, we call the inner (or *upper*, if you will) ring **conduct ring**, while the outer ring **regular ring**. Nodes in the conduct ring are called **super peers**, while nodes in the regular ring are called **regular nodes** (or **regular peers**). Note that our design philosophy can be repeatedly applied to reduce the maintenance costs of the conduct ring, yielding a hierarchical Chord (**Chord^k**) with more and more stable structures in the hierarchy (Fig. 2, right). High level rings can therefore be assigned more critical tasks that demand high stability of structure. A potential application of the hierarchical Chord is Domain Name Service (DNS). In fact, DNS has been considered as a possible application of Chord lookup service [1]. In the real world, DNS is implemented by a hierarchy of servers, where high-level servers are usually required to be more stable (and powerful) than low-level ones. Chord^k can be used to realize the deployment of the service.

In the following, we describe how to construct the two-layer system. Practically, we shall assume the existence of a Chord ring, and then attempt to construct a conduct layer dynamically from the

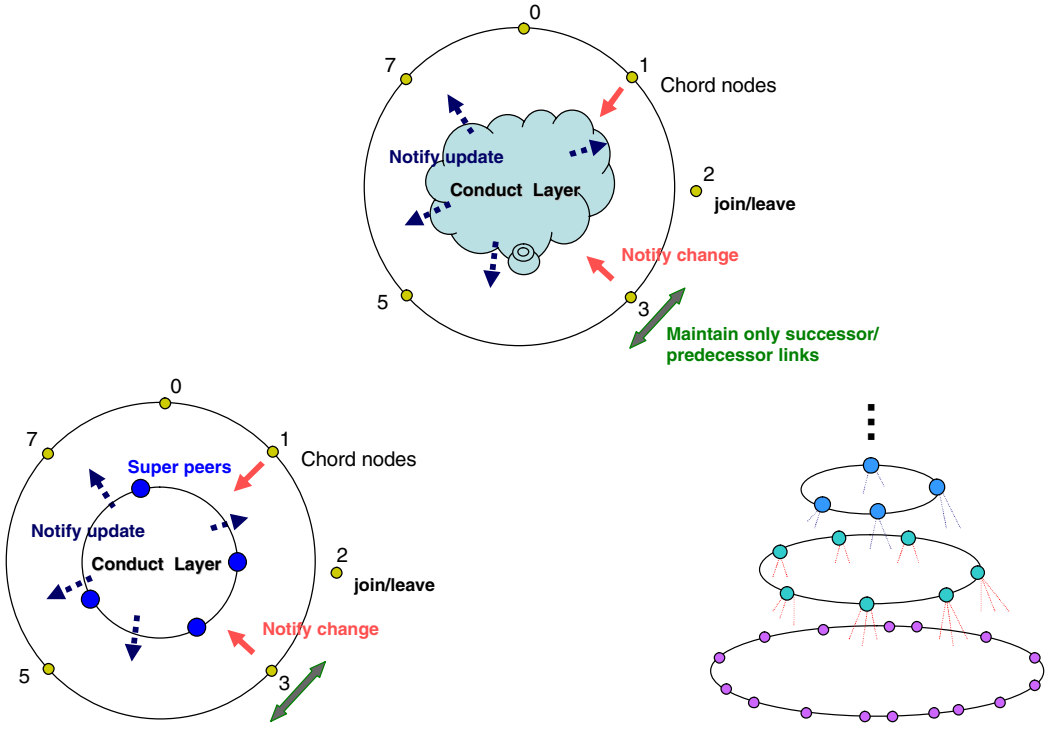

Fig. 2. Separating fingers maintenance from Chord. Top: general architecture. Left: when the conduct layer is realized by another Chord ring. Right: a hierarchical Chord.

ring. So the first problem we face is to select super peers from regular nodes.

4.1. Super peers selection

Because the conduct layer is responsible for informing nodes in the regular ring of the change (due to node joining and leaving) in the ring, it must be constructed from nodes that are relatively powerful and stable. The "power" of a node can be measured by its bandwidth, storage space, and CPU speed. Stability can be measured by the average length of continuous time that a node is available (called *session length* or *uptime*), average *downtime*, or *availability* $\left(\frac{\text{uptime}}{\text{uptime}+\text{downtime}}\right)$. Depending on the application, qualification of super peers can be measured as a combination of the above factors by using a scoring function that gives different weights to them. For simplicity, in the paper we take only uptime into account.

To decide whether a node is qualified for being super peer, every node records each session length it has stayed in the system. After some number of uptimes have been recorded, an average is taken and randomly distributed to other nodes a node has contacted. Note that the uptime information can be piggybacked on lookup messages to avoid generating extra traffic to the system. As time goes by, more and more node uptimes have been collected by each node. Based on the collection, a node can estimate whether its uptime is on some top percent of the system, or recommends some node as super peer. For example, if a node n can become a super peer only if its uptime is no less than all other 30 uptimes it has collected, then the probability that n is not on the top 10% rank is $(1-0.1)^{30} \approx 0.042$.

In the above we have assumed that peers are trustable. A malicious peer may unfaithfully report its uptime, or directly claim itself as super peer. To cope with this, we can let the uptime be observed and/or verified collaboratively by its neighbors, and let super peers be elected by other peers that have collected their uptimes. Since Chord neighborhood is decided by hashing node IP/port, it is difficult for a number of peers to collude to place themselves in a consecutive segment of the Chord ring. If one wishes, a more complex trust and reputation mechanism can be introduced to elect super peers (e.g., [23–25]).

4.2. Conduct layer construction

Once a node is promoted to be a super peer, it needs to find another super peer as a hook up point to join into the conduct layer. The challenge of the problem lies in that during the conduct layer construction stage, super peers may emerge concurrently. In the absence of any centralized mechanism, multiple and disjointed conduct rings could be created simultaneously, resulting a partition in the conduct layer.

To solve the problem, we could let new super peers search along the regular ring to find an existing super peer. This, however, is time consuming and requires $O(N)$ steps. Instead, we construct a binary tree over the regular ring, as shown in Fig. 3. Every node stores the identifier of some (arbitrary) super peer found in its right subtree. So a query for a super peer can be resolved by forwarding the query up to the root. For example, assume y is a super peer in the right subtree of z. Then when a new super peer x looks up for an existing super peer, its query can be resolved by z in three hops. While a query is being resolved, the internal nodes in the path from leaf to root can also update their information about the super peers they have seen. For example, when x's query is forwarded to z, node w will learn that x is the first super peer in its right subtree. x's query stops at the node (z in this example) when it encounters a super peer.

The search of a super peer then requires at most $O(\log N)$ links to traverse in the binary tree. If we can make each link in the binary tree correspond to a direct hop in the Chord ring, then searching for a super peer can be done in at most $O(\log N)$ hops in the ring. To do so, we adopt the method proposed by Karger and Ruhl [26] to embed the tree into the Chord regular ring. Assume that the Chord id space is normalized into $[0,1]$. Let a_0 be a base id, and let $\langle a,b\rangle$ denote the identifier $(a_0+\frac{b}{2^a})$ mod 1 for $0 \leqslant a < m$ and $0 \leqslant b < 2^a$, where $a,b \in Z$. Note that $\langle a,b\rangle$ and $\langle a+1,2b\rangle$ actually represent the same node (because $\left[a_0+\frac{b}{2^a}\right] = \left[a_0+\frac{2b}{2^{a+1}}\right]$), although they will be drawn as two separate nodes in the tree. We let node $\langle a,b\rangle$ be the parent of the two nodes $\langle a+1,2b-1\rangle$ and $\langle a+1,2b\rangle$. Fig. 4 left illustrates a search tree constructed from eight nodes.

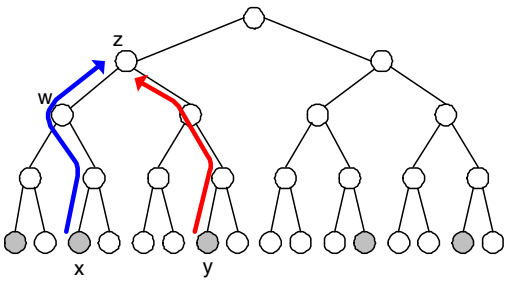

Fig. 3. A binary search tree for super peer discovery. Gray nodes represent super peers.

When embedding the search tree into a Chord ring (see Fig. 4), forwarding a query from left child $\langle a+1,2b-1\rangle$ to its parent $\langle a,b\rangle$ in the search tree corresponds to a one hop jump in the ring. This is because the id distance between the two nodes is $\frac{1}{2^{a+1}}$, and fingers at each node in a Chord ring respectively, jump $2^{-m}, 2^{-m+1},\ldots,2^{-1}$ distance away. Forwarding a query from a node's right child to the node costs nothing as they are the same node. Since the tree has height $\log N$, it takes at most $\log N$ hops for a node to find a super peer in the Chord ring. If a node cannot find any super peer, then it must be the first super peer in the system. In this case it creates a conduct ring consisting of itself, waiting for other super peers to join.

We note that once the base id a_0 is chosen, the position of each regular node in the embedded search tree is determined. So it costs little to maintain the tree. Moreover, the search tree is used only to avoid concurrent construction of more than one conduct ring. As we shall see in the following section, after a conduct ring has been constructed, every regular node needs to know some super peer in the conduct ring. Then, a new node can learn of a super peer from its successor in the regular ring. Even if the new node later becomes super peer, it can join the conduct layer via the previous super peer without traversing along the search tree. In fact, during the construction stage, once a regular node knows some super peer, it can discard the link information it maintains for the embedded search tree. This is because all queries to the node for finding a super peer can stop there as the node can resolve them. So while the conduct layer is being constructed, the tree is also gradually abandoned by regular nodes.

A super peer joins the conduct layer using the same Chord joining procedure. A super peer has a regular Chord node id and a super peer id. For simplicity, we let super peer assume the same identifier it uses in the regular ring. The main job of super peers is to store link and finger information in the regular ring to help regular nodes maintain their ring. This will be clear in the following section when

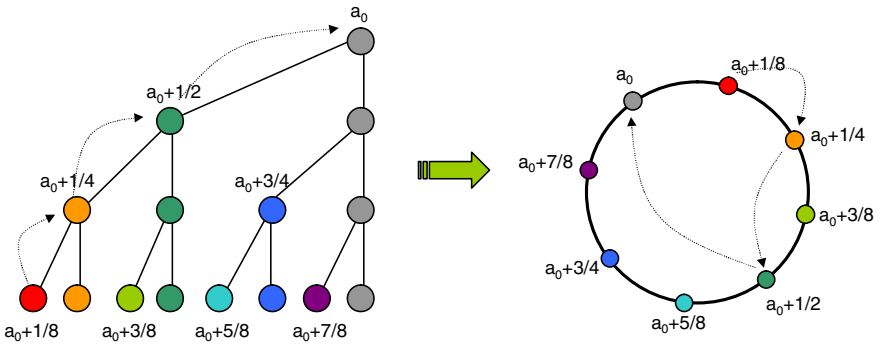

Fig. 4. Embedding binary search tree to Chord ring.

we discuss the maintenance mechanism in Chord^2. As shall also be seen shortly, unifying the identifier of a super peer with its regular role can also help reducing communication costs in the conduct ring.

4.3. Modified Chord maintenance mechanism

Like Chord, when a node n joins the system (the regular ring in Chord^2) via an existing node n', it asks n' to find n's successor and then builds its finger tables with the help of the successor. In addition, we require n to insert a ***link object*** into the conduct layer. The object consists of a key, and the successor and predecessor link information. It has the following format:

$$link\_object(n) = \langle n, n.successor, n.predecessor \rangle$$

The identifier of the owner of the link (i.e., n) is used as the key of the object.

Node n also inserts its fingers into the conduct layer. For each finger, a ***finger object*** $\langle y, i, n \rangle$ is inserted, where y is the node the finger points to, and i is the level of the finger in the finger table. In general, finger objects are of the form:

$$finger\_object(target) = \langle target, level, owner \rangle$$

Note that we use the target a finger points to as the key of the object.

The link objects and the finger objects are inserted into the conduct layer using their keys so that when a node x leaves the regular ring, some super peers can know which nodes have fingers pointing to x, and then can direct them to replace x with $x.successor$ in their finger tables. The following is the new joining procedure.

// node n joins through node n'
n.**join2**(n')

1. $predecessor := \text{nil}$;
2. $s := n'.find\_successor(n)$;
3. $successor := s$;
4. $build\_fingers(s)$;
5. // find super peer responsible for n via s
6. $x := s.Superpeer(n)$;
7. // insert link object into super peer
8. $x.insert\_link\_obj(\langle n, s, \text{nil} \rangle)$;
9. // insert finger objects into super peers responsible for them
10. **for** $i := 1$ **to** $m - 1$
11. $\quad x.insert\_finger\_obj(\langle finger[i], i, n \rangle)$;

Note that in line 8, node n needs to know the super peer responsible for the link object (which has key n) it wishes to insert. For each key k, we use $Superpeer(k)$ to denote the super peer responsible for the key, and refer to the super peer as *the super peer of* k. In the joining process, node n finds its super peer via its successor (line 6). After it finds its super peer, n can maintain a link to the super peer so that subsequent communication with the super peer takes only one hop. Also note that since super peers use the same ids as their regular node ids, and since a new node n cannot join into the conduct ring before it has completed the joining operation in the regular ring, $Superpeer(n)$ must be the same as $Superpeer(s)$ in line 6. So $Superpeer(n)$ can be obtained from s without a lookup operation in the conduct ring.

For the insert of the finger objects in line 11, n can locate their super peers via n's super peer. Each insert will then take at most $\log C$ hops in the conduct ring, where C is the size of the conduct ring.

In Chord^2, each node n still executes the stabilization procedure in Chord periodically to maintain successor and predecessor links. The change of the links must be caused by node joining or leaving.

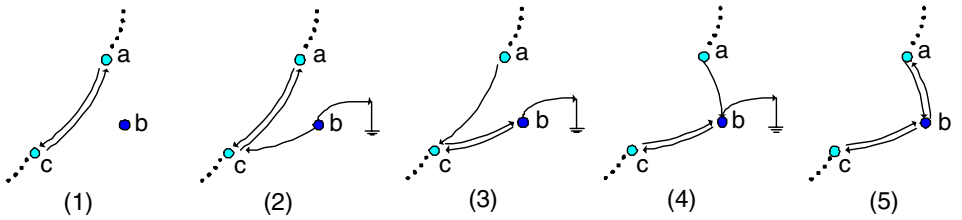

Fig. 5. Node joining process in Chord. (1) Node b wishes to join. (2) After b.*join*(). (3) After b.(c.*notify*()). (4) After a.*stabilization*(). (5) After a.(b.*notify*()).

Recall the node joining process in Chord (see Section 3.1). In Fig. 5, assume node b is joining the ring in between nodes a and c. After executing b.*join*(), b's successor is set to c, and its predecessor is set to nil; but the successor/predecessor links of a and c are not yet affected. When b executes *stabilization*(), it will call c.*notify*() to ask c to update c's predecessor. Node c can therefore learn of b's joining. After c has set its predecessor to b, an execution of *stabilization*() by a will cause a to find that b is sitting in between a and c, and so will also learn of b's joining and update its successor link to b. Then, a will call b.*notify*() to ask b to update its predecessor to a. The links between a, b, and c are then fixed.

From the above, we see that both a and c can detect the joining of b, but it suffices for c to do so. So in Chord[2], a node n detects node joining if it finds that a new node should be placed in between its current predecessor and n. On the other hand, the exit of a node x can be detected by x's predecessor if the predecessor observes that its new successor is sitting after x. The following is the new stabilization procedure.

```
// node n periodically probes its successor link
n.stabilization2()
1.  Succ_old := successor;
2.  check_predecessor();
3.  n.fix_successor(); // check if successor is alive
4.  x := successor.predecessor;
5.  if (x ∈ (n, successor))
6.      successor := x;
7.  successor.notify2(n);
8.  Succ_new := successor;
9.  if Succ_new ≠ Succ_old
10.     Superpeer(n).link_update(⟨n, Succ_new,
        predecessor⟩);
11. if Succ_old ∈ (n, Succ_new)
12.     Superpeer(Succ_old).notifyLeave (Succ_old,
        Succ_new);

// n' notifies n to be n's predecessor
n.notify2(n')
1. Pred_old := predecessor;
2. if n' ≠ Pred_old
3.    Superpeer(n).link_update(⟨n, successor, n'⟩);
4. if (predecessor = nil or n' ∈ (predecessor, n))
5.    predecessor := n';
6.    Superpeer(n).notifyJoin(Pred_old, n');

// notify Superpeer(n) the joining of Pred_new
// with Pred_old being the old predecessor of n,
// where Pred_new ∈ (Pred_old, n]
Superpeer(n).notifyJoin(Pred_old, Pred_new)
1. let y = Superpeer(Pred_new); // find super peer of
   Pred_new
2. for each stored finger object obj = ⟨target,
   level, owner⟩
3.      if owner + 2^level ∈ (Pred_old, Pred_new]
4.              obj := ⟨Pred_new, level, owner⟩;
5.              transfer obj to y;
6.              owner.fix_finger(level, Pred_new);

// notify node n to fix its ith finger
n.fix_finger(i, target)
1. finger[i] := target;

// notify super peer x the leave of Succ_old, with
// Succ_new being the new successor of Succ_old
x.notifyLeave(Succ_old, Succ_new)
1. // remove link object of key Succ_old
2. remove link_object(Succ_old);
3. y = Superpeer(Succ_new); // find super peer of
   Succ_new
4. for each stored finger object obj = ⟨target,
   level, owner⟩
5.    if target = Succ_old
6.          obj := ⟨Succ_new, level, owner⟩;
7.          owner.fix_finger(level, target);
```

In the procedure n.notify2(n'), when node n finds that a new node n' has joined the system and

become n's predecessor, n updates its link object in its super peer (line 3). Then n notifies its super peer to check its finger objects. This is done by procedure $Superpeer(n)$.notifyJoin($Pred_{old}, n'$). The finger objects are examined as follows: For each object $\langle y, i, k\rangle$ (meaning that k's ith finger points to y), if $k + 2^i \in (Pred_{old}, n']$, the object should be changed to $\langle n', i, k\rangle$, and be transferred to the super peer of n'. Moreover, the regular node k should also be notified to change its ith finger to n' (line 6). Note that when y transfers a finger object to the super peer of n' (line 5 of notifyJoin), the super peer of n' must either be the same as $Superpeer(n)$, or n' itself. This is because super peers assume the same identifiers they use as regular nodes. So before n' joins the system, there is no (alive) peer in between $Pred_{old}$ and n. So for any $i \in (Pred_{old}, n]$, $Superpeer(i) = Superpeer(n)$. If we do not allow a new joining peer to act as a super peer right after it joins into the regular ring, then n' cannot be a super peer when line 1 of notifyJoin is executed. So it takes no communication costs to locate the super peer of n' from $Superpeer(n)$ and to transfer the objects.

To illustrate the node joining process, consider Fig. 6. Assume that node 15 has joined the system, and node 17 detects this by finding that its predecessor has changed from 13 to 15. Node 17 reports 15's joining to its super peer (SID 20) and updates its link object to $(17, 18, 15)$. Then, super peer 20 checks its finger objects to see if they need to be updated. The finger object $(17, 2, 11)$ needs to be changed to $(15, 2, 11)$ because $11 + 2^2 \leqslant 15 < 17$. Similarly, $(17, 1, 13)$ needs to be changed to $(15, 1, 13)$. Super peer 20 then notifies nodes 11 and 13 to refresh their fingers. Note that when super peer 20 learns that 15 has joined the regular ring, it can also notify both 13 and 15 to fix their successor and predecessor links, rather than letting them fix the links when they execute the stabilization procedure.

On the other hand, node leaving is detected when a node n finds that a node $Succ_{new}$ sitting after its old successor $Succ_{old}$ has become its new successor (line 11 of *stabilization*2()). In addition to updating n's own link object, n also notifies the super peer of $Succ_{old}$, say x, the absence of $Succ_{old}$. In the procedure $x.notifyLeave()$, x checks if it has any finger object with target $Succ_{old}$. If so, the target must be changed to $Succ_{new}$, and the owner should also be notified to update its finger. Note that the modified finger objects must also be handled by the same super peer, and so there is no need to transfer the objects. This is because there is no node in between $Succ_{old}$ and $Succ_{new}$, and so the super peer of $Succ_{old}$ and the super peer of $Succ_{new}$ must be the same.

To illustrate the node leaving procedure, consider Fig. 7. Assume that node 17 has left the Chord ring, and node 15 observes the departure of 17 by finding that its successor has changed to 18. Node 15 informs its super peer (SID 20) to update its link object, and the super peer of node 17 (also SID 20) to delete node 17's link object as well as to update its finger objects. The finger objects $(17, 3, 8)$ and $(17, 2, 13)$ in SID 20 need to be changed to $(18, 3, 8)$ and $(18, 2, 13)$, respectively. Finally, node 8 is notified to change its 3rd finger to 18, and similarly node 13 is notified to change its 2nd finger to 18.

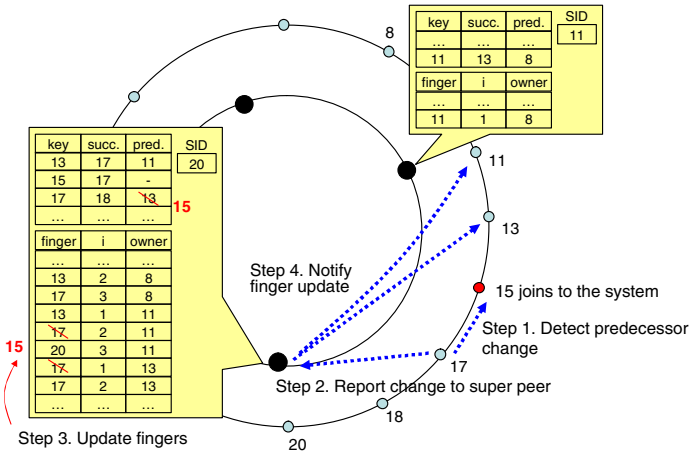

Fig. 6. An illustration of node joining in Chord[2].

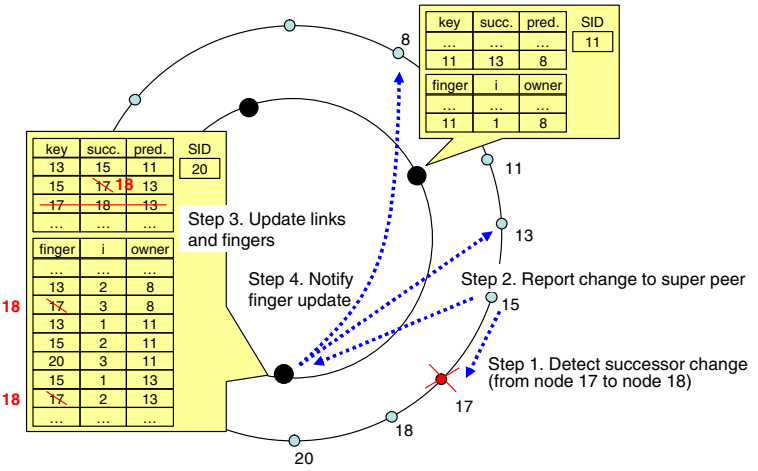

Fig. 7. An illustration of node leaving in Chord[2].

In the above we illustrated how finger objects were used to maintain the regular ring. Link objects stored at the conduct layer help super peers know the neighboring relation of regular nodes, and therefore know how to correct finger objects. For example, in Fig. 7 super peer 20 knows that node 18 is the successor of node 17, and so knows that fingers pointing to 17 should be corrected to pointing to 18 if node 17 fails. In addition, the link information stored at a super peer provides the super peer a view of the identifier space for some segment of the regular ring. Collectively, the super peers have a "macro view" of the regular ring so that lookup in the regular ring can also be performed via the conduct ring. For example, in Fig. 8, lookup of *successor*(17) from node 2 in the regular ring can be performed by first routing the message in the conduct ring to *Superpeer*(17), which is SID 20. Then, since SID 20 has the link information from nodes 14 to 20, it knows where *successor*(17) should be. Lookup in the conduct ring takes $O(\log C)$ hops, as opposed to $O(\log N)$ in the regular ring, where C and N are the sizes of the two rings, respectively. If super peers are also selected from high speed and high bandwidth nodes as in Brocade [10], then lookup via the conduct ring not only takes fewer hops, but each hop also takes less time.

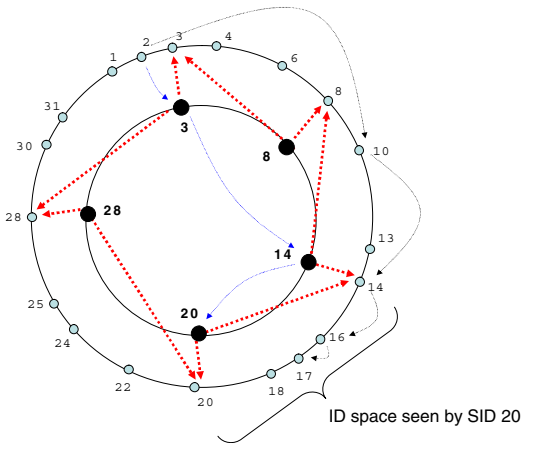

Fig. 8. Lookup of *successor*(17) from node 2 via the regular ring (which takes 4 hops) vs. via the conduct ring (which takes only 2 hops plus the contact between node 2 and its super peer).

4.4. Maintenance of the conduct layer and fault tolerance

Since the conduct ring is itself a Chord ring, it can be maintained by the regular way a Chord ring is maintained. Moreover, although super peers are relatively stable, failures of them cannot be excluded. As failures of super peers may cause lost of link and finger objects, to increase fault tolerance, we backup a super peer's objects to its successor. This backup process can be incorporated into the stabilization process by piggybacking object information on the stabilization messages to avoid additional message costs. When a super peer x fails, its successor y will become the super peer responsible for all objects that were handled by x, as by definition *successor*(x) now equals to y. So the owners of the objects do not need to re-insert their objects—which would otherwise be a tedious process. Higher degree of fault tolerance can be obtained by replicating a super peer's objects to a number of consecutive successors.

In an untrusted environment, we also need to ensure the integrity of the information stored at super peers. The problem may occur if some malicious peer is selected as super peer, or attempts to pollute the information with incorrect link/finger objects. For the former, as discussed in Section 4.1, some trust and reputation mechanism can be introduced to prevent malicious peers from becoming super peers. For the latter, super peers may verify reports of node joining/failure by attempting to contact the nodes concerned.

4.5. Maintenance overhead

We now calculate the overall maintenance costs in Chord^2. We assume that there are N nodes in the regular ring, among which C nodes are selected as super peers. Consider first the node joining process. When a node n joins the regular ring, we need to build a finger table for n, and then to insert its link and finger objects into super peers. There are $O(\log N)$ fingers. If we use the lookup procedure in the regular ring to find the target of each finger, then the total costs for building a finger table would be $O(\log^2 N)$—same as in Chord. However, as commented at the end of Section 4.3, we can look up the targets in the conduct ring. For each target *successor*$(n+2^i)$ of the ith finger, we can find the super peer of $n+2^i$ to resolve the lookup. Lookup in the conduct ring takes $O(\log C)$ messages. So the cost to build a finger table can be reduced to $O(\log C \cdot \log N)$. The cost to insert n's link and finger objects is also $O(\log C \cdot \log N)$.

Moreover, when node n joins, its successor will detect its joining and inform the super peer of n in the conduct ring. As noted in Section 4.3, it takes at most one hop (and so one message) to locate n's super peer from the super peer of n's successor, as well as to transfer finger objects. n's super peer then has to notify at most $\log N$ regular nodes to update their fingers to n. So $O(\log N)$ messages are generated to update fingers due to n's joining. All together, a node joining to the system costs $O(\log C \cdot \log N)$.

When a regular node n leaves Chord^2, either it will inform its super peer, or its predecessor will detect its departure. Like node joining, it also costs $O(\log N)$ messages to update fingers in the regular ring.

In contrast, the task of fixing successor/predecessor links and fingers in Chord is done via the periodical execution of *stabilization*() and *fix_fingers*(), regardless of whether the network changes or not. In Chord^2, regular nodes must also periodically execute a similar *stabilization2*() to maintain successor/predecessor links, but the process to fix fingers, as analyzed above, is notified passively by super peers, and only when change occurs. A round of *stabilization*() and *stabilization2*() costs $O(1)$ messages, while a round of *fix_fingers*() costs $O(\log^2 N)$ messages. So Chord^2 costs only $O(1)$ messages to maintain the regular ring, as compared to $O(\log^2 N)$ in Chord.

We have not yet considered the cost to maintain the conduct ring. Since the conduct ring is maintained as a regular Chord, a super peer joining to the ring costs $O(\log^2 C)$ messages, and periodically it has to spend $O(\log^2 C)$ messages to maintain successor/predecessor links and fingers. In addition, the joining and leaving of super peers also cause link and finger objects to be moved between neighboring super peers. In average, each super peer handles $O\left(\frac{N \log N}{C}\right)$ fingers and $O\left(\frac{N}{C}\right)$ links. For a typical value of N and N/C ratio in the order of dozens to hundreds, the object information can be packed into a number of packets. So the additional message cost is $O(1)$. Note that the above did not consider the case that link/object information is lost due to super peer leavings or failures. As discussed in Section 4.4, we backup a super peer's objects to its successor so that when the super peer fails, regular nodes do not need to re-insert their objects to the super peer's successor. Otherwise, a super peer leaving will cost an additional $O\left(\frac{N \log N}{C}\right)$ messages to update object information.

Table 1 summarizes the comparison. From the table we see that Chord^2 can reduce the maintenance costs in Chord by moving the periodical execution of *fix_fingers*() from the regular ring (which costs $O(\log^2 N)$) to the conduct ring (which costs $O(\log^2 C)$). The merit of Chord^2 is further manifested by reducing the periodical maintenance frequency in the conduct ring, as it is composed of relatively more stable peers than the regular Chord. Furthermore, as we shall see in the following

Table 1
Maintenance overhead of Chord and Chord^2

	Chord	Chord^2
Join	$O(\log^2 N)$	$O(\log C \cdot \log N)$
Leave	–	$O(\log N)$
Periodical maintenance	$O(\log^2 N)$	$O(\log^2 C)$... for super peers $O(1)$... for regular nodes

section, the stability of the super peers allows us to introduce a simpler and more cost-effective finger maintenance in the conduct ring, which can further reduce the maintenance costs from $O(\log^2 C)$ to $O(\log C)$.

Finally, we comment on the load added to super peers, which in turn affects the decision on determining the conduct ring size C. For storage, recall that in average each super peer handles $O(\frac{N \log N}{C})$ fingers and $O(\frac{N}{C})$ links. For bandwidth, since the joining and leaving of a regular node generates $O(\log C \log N) + O(\log N)$ messages to the conduct layer, in average each super peer has to sustain $\lambda \cdot O(\frac{\log C \log N}{C})$ traffic, where λ is the node arriving/leaving rate. So if C is relatively small compared to N, then bandwidth and CPU power must be taken into account and given some significant weight when electing super peers.

4.6. Further improvement

We have learned that finger tables account for the major maintenance overhead, and updates to them can be linked with the successor stabilization process. Based on this, another way to reduce Chord maintenance costs is to remove *fix_fingers*(), and let the predecessor of a finger's target be responsible for maintenance of the finger. More precisely, recall from Section 4.3 that a finger object is of the form

$$\langle target, level, owner \rangle$$

Let x be the target of a finger f. Then, instead of employing super peers, we can store a copy of f at the predecessor (say y) of *successor*(x). When the predecessor detects a change of its successor that may cause the change of *successor*(x), it informs the owner of f to update the finger's new target. The predecessor y may also need to hand over its responsibility of f to its new successor. As such, the $O(\log^2 N)$ maintenance costs resulted from *fix_fingers*() can be removed, leaving only the periodical invocation of *stabilization*().

In the above modification, fingers pointing to a node x rely on x's immediate predecessor to maintain them. As the predecessor may also leave the system, to increase fault tolerance, we may need a list of l predecessors to maintain the fingers so that the owners of the fingers do not need to re-insert them unless all the l predecessors have failed simultaneously. One can see that if many nodes in the system have short uptime, then a large l may be required, thereby offsetting the benefit gained by removing the regular *fix_fingers*() procedure. However, if all nodes are relatively stable as those selected for the conduct ring in Chord^2, then adopting this scheme can further reduce the maintenance costs.

5. Experimental results

We develop an event-driven simulator written in Java to evaluate both Chord and Chord^2. We use empirical values to simulate peer joining and leaving activities. Specifically, Saroiu et al. [9] 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. 9, and a similar distribution has also been observed in [27]. The original data was observed in a 12-h interval. 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 peer 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.

Initially, we let 10,240 nodes concurrently join the network. Each node n stays in the network according to the session length δ it has chosen. After n has left the network, a new node will join the network after some random time t. To keep the network size roughly stable, t follows exponential distribution with mean δ. Note that we have also verified with different network sizes finding similar results, so here we report only the case of 10,240 nodes.

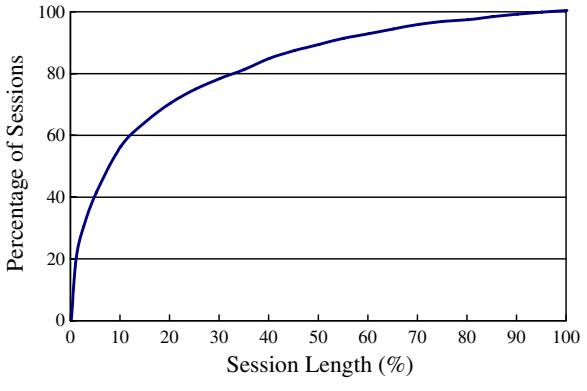

Fig. 9. CDF of system session lengths.

Given the CDF distribution in Fig. 9, we can calculate the system average session length, denoted by $\overline{\delta_T}$. According to the CDF, $\overline{\delta_T} = 0.195T$. In the simulation we will scale T so that $\overline{\delta_T}$ ranges in between 10 min and 2 h to model the dynamics of P2P systems. All measurements of the system performance will be with respect to different $\overline{\delta_T}$.

When simulating Chord[2], super peers are selected from nodes whose session lengths are on the top 15%, 10%, and 5% of the system, respectively. For example, in Fig. 9, where the system average session length is $\overline{\delta_T} = 0.195T$, the average session length of the top 15%, 10%, and 5% peers are $0.651T$, $0.734T$, and $0.847T$, respectively. To simplify the simulation, we assume a peer knows the global distribution and so can determine if it is qualified for acting as a super peer. In practice, as discussed in Section 4.1, we can let peers distribute their average session lengths along with lookup queries to let them estimate their ranks.

The system performance will be measured in terms of lookup success rate and lookup efficiency. Lookup operations are generated independently by each (alive) node according to Poisson distribution with waiting time $\lambda = 30$ s; that is, each node initiates a lookup (with a random target) every 30 s on average. In addition, we also measure maintenance overhead, which is calculated by the number of maintenance messages transferred per node per minute. Note that only messages generated due to maintenance (e.g., for stabilization and fixing fingers) will be counted; these messages will be distinguished from the messages caused by the previous lookups randomly generated at each node.

In the simulation each node leaves the system without notifying its neighbors, and therefore other nodes become aware of the leave only when they execute the stabilization procedure. Recall that a lookup is considered failed if and only if it is forwarded to the successor node which has left but the relay node is not aware of the leave. When a node forwards a lookup message to its ith finger node but the finger node has failed, the lookup message will be re-transferred to its $(i-1)$th finger node. In case there are lots of stale fingers, lookups will route through more hops than would be normally required (i.e., $O(\log N)$ steps). If fingers are fixed promptly, average hops for lookup will approximate to a stable value in the static network condition. In our hypothesis, the maintenance efficiency may therefore be demonstrated with the number of hops traversed per lookup, i.e., lookup efficiency.

The following two parameters in maintenance will be used:

- **S_interval:** the interval for each node to check its successor link via the stabilization procedure.
- **F_interval:** the interval for each node to refresh one of its fingers via the *fix_fingers*() procedure.

5.1. Lookup success rate and lookup efficiency in Chord

In the first experiment, we study how lookup success rate in Chord is affected by the two maintenance intervals S_interval and F_interval. For each chosen S_interval and F_interval, we measure the lookup success rate with respect to the system average session length $\overline{\delta_T}$. For each fixed $\overline{\delta_T}$, we conduct five independent runs of simulation, each of which lasts for 4 h. In each run, about 4,400,000 samples of lookups are collected. The success rate is measured by the proportion of successful lookups to all lookups collected. The final lookup success rate is the average of the lookup success rate in each run. Because the sample size is very large, the measured values between different runs usually differ in less than 0.2%.[4] The results of the experiment are shown in Fig. 10. On the left chart, we fix F_interval=30 s, and vary S_interval from 10 to 60 s. On the right, we fix S_interval=30 s but vary F_interval.

Since lookup success rate depends on how effective successor links are fixed, shorter S_interval yields higher success rate. On the other hand, F_interval has little effect on the success rate. This is because during the lookup process, if a node forwards a lookup message to its ith finger node but the node has failed, then the $(i-1)$th finger will be used, and so on until the zeroth finger—the successor link—has been reached. So lookup success rate basically relies on how effective successor links are maintained. From the figure one may also note that F_interval slightly affects the success rate when nodes have short session lengths. This is because when nodes are very unstable, many successor links are stale (given a fixed S_interval). So the probability of encountering a failed node increases during the lookup if the lookup hop count increases (after

[4] Thus, each result shown here represents a sample mean within a confidence interval (±0.1%) given by a Student's t-distribution with 4 degree of freedom and at 99% confidence level.

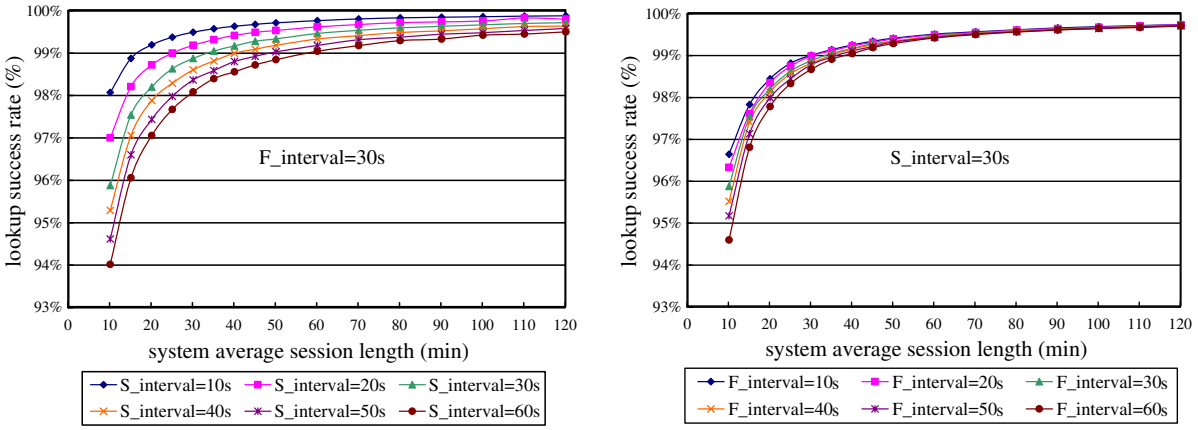

Fig. 10. S_interval (left) and F_interval (right) vs. lookup success rate in Chord.

all, in the worst case lookup needs to traverse through successor links to the destination). So if fingers are fixed slowly, then more hops will be needed to route a lookup message to a destination, and so the success rate of the lookup will also decrease.

Although stale fingers have little effect on success rate, they may decrease lookup efficiency (in terms of the number of hops required per lookup). In a static network, a lookup in Chord takes $\frac{1}{2}\log N$ hops on average. If fingers are stale, then a halfway style jump to the destination may be replaced by more jumps of short steps. So it would take more than $\frac{1}{2}\log N$ hops on average per lookup. In the second experiment, we study how lookup efficiency is affected by F_interval and S_interval. The experiment method is similar to the previous one. The lookup hop count is measured by taking, for a given setting of F_interval and S_interval, the average lookup hop count of all lookups issued in the simulation. The results are shown in Fig. 11. They confirm that F_interval does affect the average lookup hop count, especially when nodes join/leave frequently. When the system is relatively stable (the system average session length is long), the average hop count is about 7. This result matches the theoretical analysis that the expected number of hops for halfway routing is $\frac{1}{2}\log_2 N = \frac{1}{2}\log_2 10240 \cong 6.66$. On the other hand, S_interval has little effect on lookup efficiency, as successor links are used only for the last hop.

5.2. Chord vs. Chord^2

We then study the lookup success rate and lookup efficiency of Chord^2. The main purpose of the study is to find a setting for Chord^2 so that both

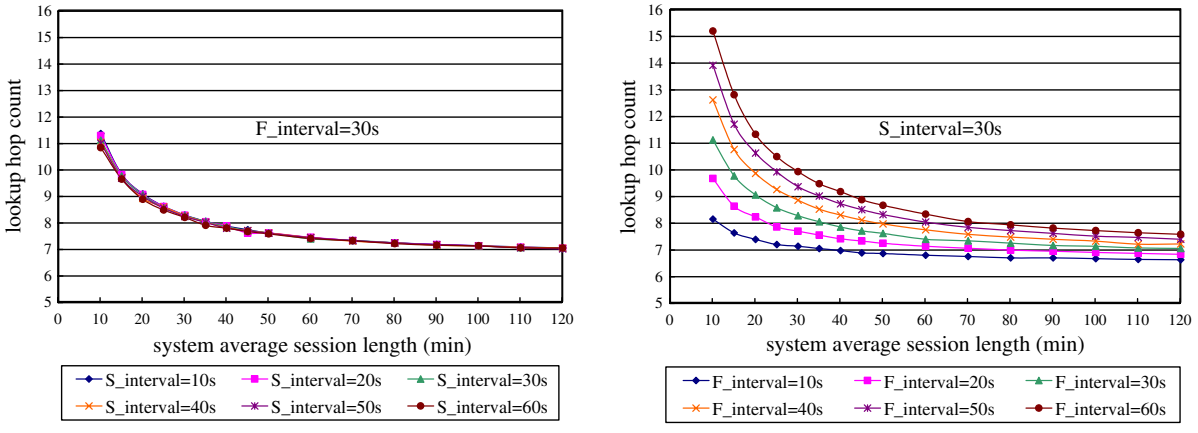

Fig. 11. S_interval (left) and F_interval (right) vs. lookup hop count in Chord.

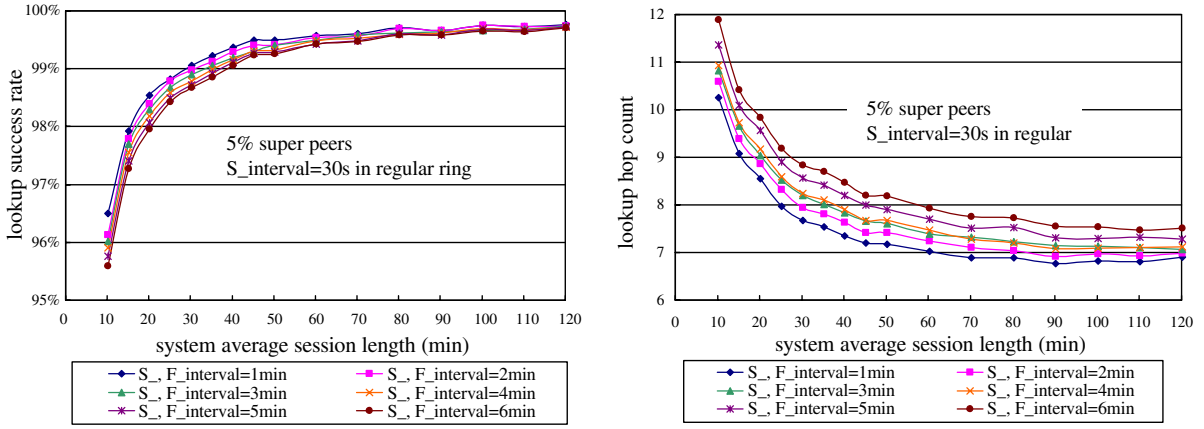

Fig. 12. Maintenance intervals of the conduct ring vs. lookup success rate (left) and lookup hop count (right) in Chord^2.

Chord and Chord^2 have the same performance. This setting then provides us the ground for a fair comparison of the maintenance overhead in the two systems. We set S_interval to 30 s for the regular ring in Chord^2, and study how the maintenance intervals in the conduct ring affect the lookup success rate and lookup efficiency of Chord^2. Super peers are selected from nodes whose average session lengths are on the top 5%. Since super peers are relatively stable, the maintenance intervals S_interval and F_interval for the conduct ring can be set longer than the regular ring. We let the two intervals have the same value, and vary them from 1 to 6 min. The results are shown in Fig. 12.[5]

From the figure we see that the shorter the intervals, the better the performance. However, the difference is not much. To explain this, recall that super peers store link and finger objects of regular nodes and are responsible for notifying them to update their fingers. So the failure of a super peer will affect some regular nodes to maintain fresh fingers. However, also recall that we backup link and finger objects of a super peer to its successor. So a regular node n can lose its objects only when (1) both its super peer x and the successor of x fail simultaneously, or (2) after n has inserted its objects to its super peer x, x fails before it replicates n's objects to its successor (x replicates the objects when it executes stabilization to check its successor link). Case (1) is unlikely to occur. For case (2), the shorter the maintenance intervals, the lower the chance for the case to occur. But since super peers are very stable, varying the intervals in a scale of 1 min yields only a small impact to the occurrence of case (2). The lost of n's objects will block it from updating its fingers when the network changes, thereby increasing the overall lookup hop count of the system. Furthermore, recall from Section 5.1 that more hop count yields a slightly higher chance for a lookup to encounter a broken successor link en route to the destination. So long maintenance intervals in the conduct ring indirectly (and slightly) affect lookup successor rate of the regular ring.

Of the different settings, we find that setting S_interval and F_interval to 3 min in the conduct ring yields similar performance to Chord (in which the S_interval is set to 30 s as in the regular ring of Chord^2, while the F_interval is also set to 30 s). In Fig. 13 we put the two systems together for ease of comparison. In addition, we also put the results for Chord^2 where super peers are selected from top 10% and top 15% nodes, respectively (with respect to their average session lengths). From Fig. 13 we see that there is little difference between the three settings 5%, 10%, and 15%. To explain this, recall from the above discussion that selecting super peers that are more stable in the conduct ring reduces the chance for regular nodes to lose their link and finger objects. So intuitively, the 5% case should yield better performance than the other two cases. However, higher quality also implies lower quantity. So there are fewer super peers in the conduct ring in the 5% case than in the other cases. As a result, each super peer in the 5% case is responsible for more regular nodes. So a failure of a super peer will block more

[5] Again, unless stated otherwise, all experiments in the simulation have been repeated five times, and the average is taken as the final result.

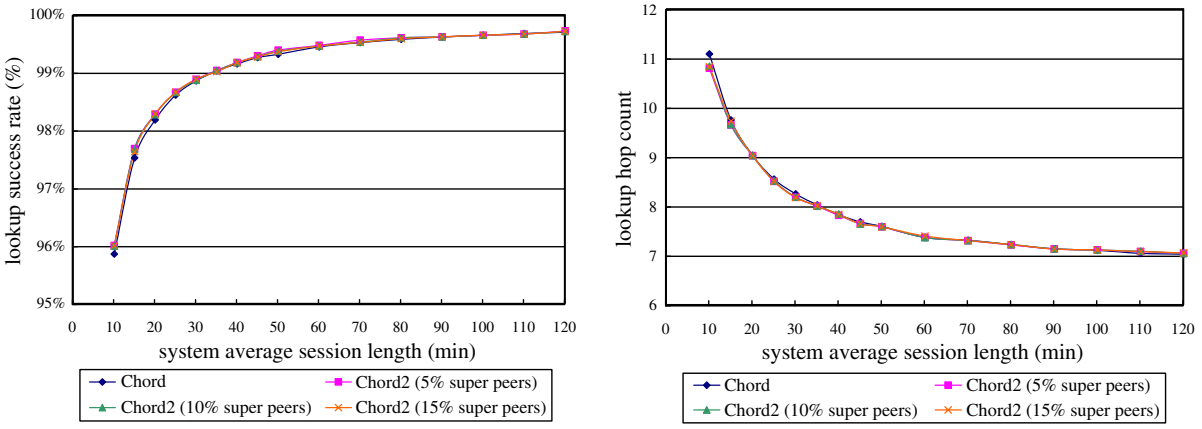

Fig. 13. Chord^2 vs. Chord in lookup success rate (left) and lookup hop count (right).

regular nodes from keeping pace with the network changes. So selecting fewer super peers in the conduct ring yields worse performance. Combining both advantages and disadvantages of selecting better quality of super peers, we have that the three cases 5%, 10%, and 15% yield roughly the same performance for Chord^2 (but notice that selecting few super peers yields less maintenance costs overall; see the following section).

5.3. Maintenance overhead

We now measure the maintenance overhead reduced by Chord^2 as opposed to the ordinary single layer structure of Chord. Based on the studies in the previous section, we shall set the maintenance intervals of the two systems as follows: the S_interval and F_interval of Chord are set to 30 s, while the S_interval in the regular ring of Chord^2 is also set to 30 s, but the S_interval and the F_interval of the conduct ring are set to 3 min. These settings cause the two systems to yield similar performance, thereby allowing us to measure the costs reduced by employing a conduct ring to maintain the regular ring in Chord without compromising the performance.

The maintenance costs are measured by the number of messages generated to maintain the system. We divide the costs into two types: for stabilizing successor links and for fixing fingers. The results for Chord and for the regular ring of Chord^2 are shown on the left in Fig. 14. Because S_interval in both systems is set to be 30 s, and because each execution of stabilization costs 4 messages, the cost for running stabilization is 8 messages per minute in both Chord and Chord^2 (regardless of the quality of the super peers).

For the cost in fixing fingers, recall that stale fingers in Chord cause lookup process to take more hops than needed, and therefore more messages. To fix the ith finger, a node n performs a lookup operation to find the most up-to-date successor node of $n + 2^i$. So the shorter the system average session length, the higher the message cost in lookup, and so the higher the message cost in fixing fingers. This is reflected in Fig. 14. When the system is relatively stable, fixing the ith finger costs about $\log 2^i = i$ lookup messages plus one reply. So a round of *fix_fingers*() costs about $\sum_{i \leqslant \log N-1}(i+1) = \frac{\log^2 N + \log N - 2}{2}$ messages. Because F_interval=1/2 min, and because there are $\log N - 1$ fingers, the time to complete a round of *fix_fingers*() is $1/2 \times (\log N - 1)$ min. So the message cost generated per minute for fixing fingers in a static ring is $\frac{\log^2 N + \log N - 2}{2} \times \frac{1}{1/2 \times (\log N - 1)} \approx 15.32$. This value is witnessed by the curve for Chord in Fig. 14.

In contrast, fixing fingers in Chord^2 is done by letting super peers notify regular nodes, and only when some regular node detects a change in the regular ring. So if the network is stable, the maintenance costs in fixing fingers approach zero. On the other hand, the number of messages for maintaining fingers also increases when the system average session length becomes shorter. This is because peers will then join and leave the system more frequently, thereby generating more messages to insert their link and finger objects.

The maintenance overhead for the conduct ring is shown on the right in Fig. 14. We note here that if we select fewer number of super peers (e.g., 5% vs. 15%), then the load of each super peer will also

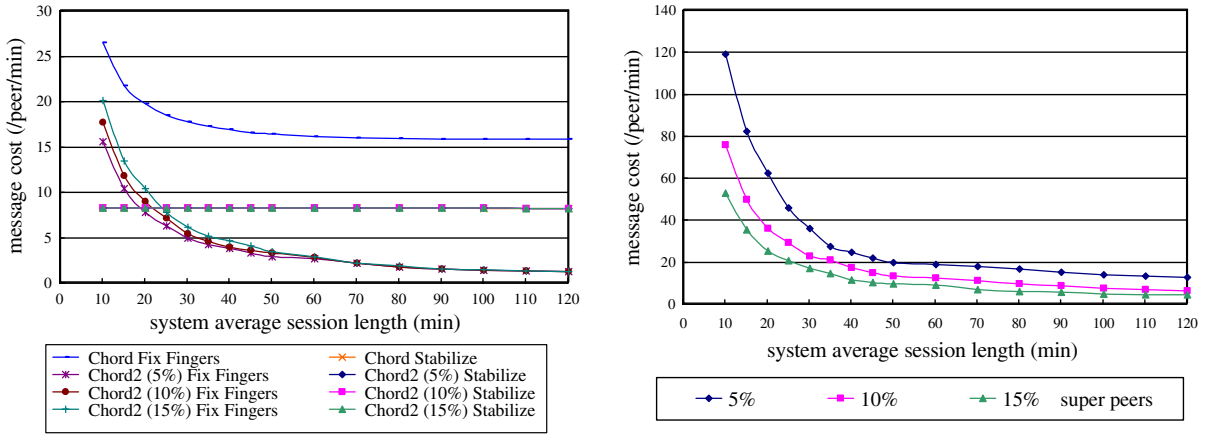

Fig. 14. Maintenance overhead in Chord^2 regular ring (left) and conduct ring (right).

increase, and thus the message count *per* super peer will be higher (but overall, the total number of messages decreases as the quality of super peers increases, as shall be seen shortly).

Taking all messages and both regular and conduct rings into account, the average number of messages generated per node per minute for Chord^2 with respect to different quality of super peers is drawn in Fig. 15. We see that better quality of super peers yields less maintenance costs, but the difference is not much, because selecting top 15% of peers to serve as super peers has already an attractive outcome. By comparing with the overall maintenance costs in Chord, we see that our two-layer structure can reduce more than half of the cost when the system average session length is more than 40 min and stabilization interval is 30 s.

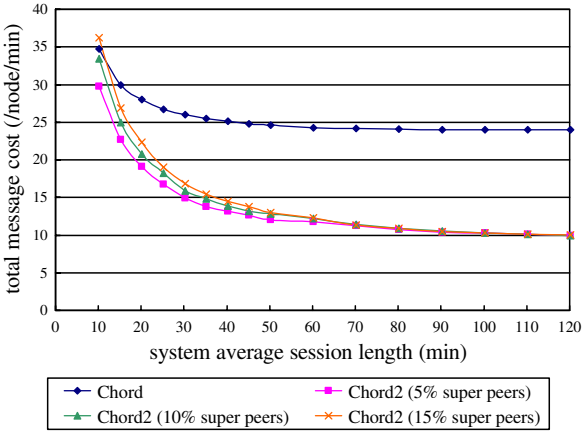

Fig. 15. Overall maintenance overhead in Chord^2 and Chord.

6. Conclusions

We have presented a two-layer structure Chord^2 to reduce maintenance costs in Chord. The system was motivated by peer heterogeneity that has commonly been observed in practical P2P systems. We select peers that are relatively powerful and stable as super peers to assist system maintenance. Regular peers now need only to periodically maintain their successor links, which costs only constant messages per execution. The maintenance of finger tables—the major overhead—now changes from a *proactive* approach to *reactive*. Because update of finger tables must be caused by peers joining/leaving the system, rather than periodically refreshing a peer's finger table, peers now update their finger tables by the notification of their super peers. Super peers in turn obtain peer joining/leaving information from regular peers when the regular peers maintain their successor links. As a result, the maintenance costs among regular peers can be significantly reduced. Although super peers also need to maintain their Chord structure, the homogeneity and stableness of them allows the structure to be maintained at much lower cost than an ordinary Chord ring would require. In short, Chord^2 reduces the maintenance costs in Chord by converting the $O(\log^2 N)$ cost per peer per maintenance interval for fixing fingers of regular nodes to $O(\log^2 C)$ for fixing fingers of super peers, where N and C are the numbers of regular peers and super peers, respectively. The merit is further manifested by the fact that the maintenance interval for super peers can be set much longer than that for regular peers as super peers are more stable than regular peers.

The hierarchical maintenance approach in Chord^2 can also be applied to other DHTs like Pastry [5], Skip graphs [28], and eCAN (conceptually a hierarchical version of CAN) [29]. In general, like Chord, overlay links in these DHTs can be distinguished into ''base links'' which connect adjacent nodes, and ''expressways'' which make each hop a larger span over the identifier space. Base links are necessary for ensuring correct routing, while expressways are auxiliary and are used to speed up routing. The number of base links per node is constant (two if the base topology is a ring, e.g., Chord, Pastry and Skip graphs; and $2d$ if the base topology is a d-dimensional Cartesian space, e.g., eCAN). In contrast, the number of expressways is typically logarithmic of the total number of nodes. So maintaining the expressways costs much higher than maintaining the base links. To reduce the cost, like Chord^2 we can divide the maintenance procedure into two parts: periodical probing of neighboring nodes by regular nodes, and, if a change is detected, the change is reported to super peers to let them inform affected nodes to fix their expressways. As such, the costly overhead in maintaining expressways is moved to super peers, and on a reactive basis. Although super peers also need to maintain their own structure, the cost is much lower as they are relatively stable.

Acknowledgements

We would like to thank the anonymous referees for their invaluable comments and suggestions. They have significantly improved the paper.

References

[1] 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.

[2] 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.

[3] P. Maymounkov, D. Mazières, Kademlia: a peer-to-peer information system based on the XOR metric, in: Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS 2002), vol. 2429 of Lecture Notes in Computer Science, Springer-Verlag, 2002, pp. 53–65.

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

[5] 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), vol. 2218 of Lecture Notes in Computer Science, Springer-Verlag, 2001, pp. 329–350.

[6] D. Malkhi, M. Naor, D. Ratajczak, Viceroy: a scalable and dynamic emulation of the butterfly, in: Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing (PODC 2002), ACM Press, 2002, pp. 183–192.

[7] 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), vol. 2009 of Lecture Notes in Computer Science, Springer-Verlag, 2001, pp. 46–66.

[8] S. Rhea, D. Geels, T. Roscoe, J. Kubiatowicz, Handling churn in a DHT, in: Proceedings of the 2004 USENIX Annual Technical Conference (USENIX'04), Boston, Massachusetts, USA, 2004, pp. 127–140.

[9] 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.

[10] B.Y. Zhao, Y. Duan, L. Huang, A.D. Joseph, J.D. Kubiatowicz, Brocade: landmark routing on overlay networks, in: Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS 2002), vol. 2429 of Lecture Notes in Computer Science, Springer-Verlag, 2002, pp. 34–44.

[11] Z. Xu, M. Mahalingam, M. Karlsson, Turning heterogeneity into an advantage in overlay routing, in: Proceedings of Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2003), 2003.

[12] The Gnutella2 Developer Network. Available from: <http://www.gnutella2.com>.

[13] B. Yang, H. Garcia-Molina, Designing a super-peer network, in: Proceedings of the Nineteenth International Conference on Data Engineering (ICDE 2003), IEEE Computer Society, 2003, pp. 49–60.

[14] R. Tian, Y. Xiong, Q. Zhang, B. Li, B.Y. Zhao, X. Li, Hybrid overlay structure based on random walks, in: Proceedings of the Fourth International Workshop on Peer-to-Peer Systems (IPTPS 2005), vol. 3640 of Lecture Notes in Computer Science, Springer-Verlag, 2005, pp. 152–162.

[15] A.T. Mizrak, Y. Cheng, V. Kumar, S. Savage, Structured superpeers: leveraging heterogeneity to provide constant-time lookup, in: Proceedings of the Third IEEE Workshop on Internet Applications (WIAPP 2003), IEEE Computer Society, 2003, pp. 104–111.

[16] L. Garcés-Erice, E.W. Biersack, K.W. Ross, P.A. Felber, G. Urvoy-Keller, Hierarchical P2P systems, in: Proceedings of the Ninth European Conference on Parallel Processing (Euro-Par 2003), vol. 2790 of Lecture Notes in Computer Science, Springer-Verlag, 2003, pp. 1230–1239.

[17] P. Ganesan, K. Gummadi, H. Garcia-Molina, Canon in G major: designing DHTs with hierarchical structure, in:

Proceedings of the Twenty-Fourth International Conference on Distributed Computing Systems (ICDCS 2004), IEEE Computer Society, 2004, pp. 263–272.
[18] A.-M. Kermarrec, L. Massouliè, A.J. Ganesh, Probabilistic reliable dissemination in large-scale systems, IEEE Transactions on Parallel and Distributed Systems 14 (3) (2003) 248–258.
[19] X.Y. Zhang, Q. Zhang, Z. Zhang, G. Song, W. Zhu, A construction of locality-aware overlay network: mOverlay and its performance, IEEE Journal on Selected Areas in Communications 22 (1) (2004) 18–28.
[20] Z. Xu, R. Min, Y. Hu, Reducing maintenance overhead in DHT based peer-to-peer algorithms, in: Proceedings of the Third International Conference on Peer-to-Peer Computing (P2P 2003), IEEE Computer Society, 2003, pp. 218–219.
[21] R. Mahajan, M. Castro, A. Rowstron, Controlling the cost of reliability in peer-to-peer overlays, in: Proceedings of Second International Workshop on Peer-to-Peer Systems (IPTPS 2003), vol. 2735 of Lecture Notes in Computer Science, Springer-Verlag, 2003, pp. 21–32.
[22] B. Leong, B. Liskov, E. Demaine, EpiChord: parallelizing the Chord lookup algorithm with reactive routing state management, in: Proceedings of the 12th IEEE International Conference on Networks (ICON 2004), vol. 1, Singapore, 2004, pp. 270–276.
[23] F. Cornelli, E. Damiani, S.D.C. di Vimercati, S. Paraboschi, P. Samarati, Choosing reputable servents in a P2P network, in: Proceedings of the 11th international conference on World Wide Web (WWW'02), ACM Press, New York, NY, USA, 2002, pp. 376–386.
[24] S.D. Kamvar, M.T. Schlosser, H. Garcia-Molina, The EigenTrust algorithm for reputation management in P2P networks, in: Proceedings of the 12th international conference on World Wide Web (WWW'03), ACM Press, New York, NY, USA, 2003, pp. 640–651.
[25] L. Xiong, L. Liu, Peertrust: supporting reputation-based trust for peer-to-peer electronic communities, IEEE Transactions on Knowledge and Data Engineering 16 (7) (2004) 843–857.
[26] D.R. Karger, M. Ruhl, Diminished chord: a protocol for heterogeneous subgroup formation in peer-to-peer networks, in: Proceedings of the Third International Workshop on Peer-to-Peer Systems (IPTPS 2004), vol. 3279 of Lecture Notes in Computer Science, Springer-Verlag, 2005, pp. 288–297.
[27] 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.
[28] J. Aspnes, G. Shah, Skip graphs, in: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2003), Society for Industrial and Applied Mathematics, 2003, pp. 384–393.
[29] Z. Xu, Z. Zhang, Building low-maintenance expressways for P2P systems, Technical Report HPL-2002-41, Hewlett-Packard Laboratories, Palo Alto, March 2002.

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.

Jiaw-Chang Wang received his B.S. and M.S. in Information Management from the National Taiwan University in 2002 and 2004, respectively. He is currently a software engineer at the Acer Co., Ltd. in Taipei, Taiwan. His main research interests are in the area of software engineering and P2P systems.