Enhancing Peer-to-Peer Systems Through Redundancy

Paola Flocchini, Amiya Nayak, *Senior Member, IEEE,* and Ming Xie

***Abstract*— Peer-to-peer systems can share the computing resources and services by directly communicating within a widely distributed network. It is important that these systems can efficiently locate, in as few hops as possible, the node storing the desired data in a large system. Thus, it is worth consuming some extra storage to obtain better routing performance. In this paper, we propose redundant strategies to improve the routing performance and data availability on Chord and De Bruijn topologies. *Hybrid-Chord* combines multiple chord rings and successors, and *Redundant D2B* maintains successors, to improve the routing performance. The proposed systems can reduce the number of lookup hops significantly (by as much as 50%) compared to the original ones, and have better fault tolerance capabilities, with a small storage overhead.**

***Index Terms*— P2P System, Chord, D2B, DHT.**

I. Introduction

PEER-TO-PEER (P2P) systems are now one of the most popular Internet applications and have become a major source of Internet traffic. Thus, it is important that these systems are scalable and can efficiently locate, in as few hops as possible, the node that stores the desired data in a large system. In other words, reducing the hop count is extremely important from the cost and performance point of view. Furthermore, nodes must be able to join and leave the system frequently without affecting the robustness or the efficiency of the system, and the load must be balanced across the available nodes.

Earlier P2P systems employ a single index server or flooding-based mechanism (Gnutella[13] and Freenet[3]) to search desired data, which are not suitable for large systems. Most latest P2P systems (e.g., [15], [12], [18], [14], [16], [10], [9], [8], [7]), use distributed hash tables (DHT) to support scalability, load balancing and fault tolerance. These systems are based on different virtual topologies, and they all employ a distributed hash table that maps names/keys to values and that is used as a supporting lookup service. DHTs manage the distribution of data among the dynamic network, and allow nodes to contact any participating node in the network to find stored resources by keys.

In P2P systems, the number of lookups for desired data is significant high, which means that locating data efficiently can save huge network communication resource. On the other hand, with the development of computer technology, local storage expense becomes negligible. Thus, it is worth consuming some extra storage to obtain efficient routing performance. In this paper, our main focus is on reducing the number of hops that are needed to locate a data item.

We present a new model for a peer-to-peer system, called *Hybrid-Chord*, to improve the routing performance and data availability of Chord[15]. Through simulations, we demonstrate the improvement of the routing performance and fault tolerance capabilities of the proposed system and compare them with the original Chord system. Here are some highlights of the Hybrid-Chord system:

- it can reduce the number of lookup hops significantly by up to 50% compared to Chord,
- it is robust and handles node failures better than Chord,
- it can always find the desired data within few hops with high probability and has better data availability than Chord,
- from scalability point of view, the total joining/leaving cost is $O(log^2 n)$, where n is the number nodes in the system, same as in Chord.

We also apply the redundant successor strategy to the D2B system of [8] (*Redundant D2B*); as for the case of Hybrid-Chord, we observe that we can significantly reduce the number of lookups improving also data availability.

The paper is organized as follows. In Section II, we describe various features of the proposed peer-to-peer system. Data lookup and routing scheme are given in Section II-C. Section II-D and II-E describe the scalability and fault tolerance issues, respectively. All Hybrid-Chord experimental results are provided in Section III followed by concluding remarks on Hybrid-Chord in Section VI. In Section IV, we propose the Redundant D2B system and describe the experimental results in Section V.

II. Hybrid-Chord System

We propose a new peer-to-peer model, called *Hybrid-Chord* or simply *Hybrid system*, which has the following two key features: multiple chord rings overlayed one on top of the other and multiple successor lists of constant size. The multiple chord rings system and the successor list system could be employed independently as a peer-to-peer model. In this paper, we describe the hybrid system obtained by combining the two ideas and resulting in an enhancement of Chord (more details can be found in [17]).

A. Multiple Chord Rings

In our system we generate a virtual network by overlaying k Chord rings one on top of the other. Our goal is to speedup data lookup and make the system more robust. The idea is

Manuscript received January 1, 2006; revised August 31, 2006.

The authors are with the School of Information Technology and Engineering, University of Ottawa, 800 King Edward, Ottawa, Ontario, K1N 6N5, Canada. Email: {flocchin,anayak,mxie}@site.uottawa.ca

Digital Object Identifier 10.1109/JSAC.2007.070103.

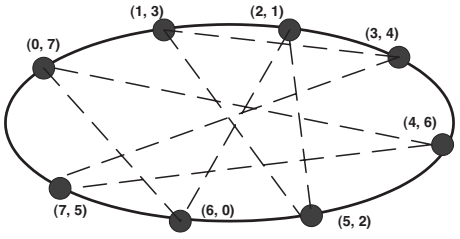

Fig. 1. 2-Chord topology

based on the fact that, if several Chord rings are overlayed, one could choose, at each step of the lookup, the best Chord ring to achieve better routing performance. Each node has k identifiers and every identifier logically corresponds to a location in one Chord. Each data item has a unique key and is mapped into the same location on different virtual Chord rings. In other words, there are k identifiers for a node, and the node is located in k logical Chord rings with different location. Since in each Chord, every data item owns only one key and is located at the same location on different Chord rings, then there are k replicas of each data item distributed in k different nodes that are in charge of its location on different Chord rings in the overlay network. We will often refer this as $k-$*Chord* model.

Figure 1 shows the example of a k-Chord topology for $k = 2$. In this figure, every node has two identifiers (Id_1, Id_2), where Id_1 is the identifier in the first Chord ring and Id_2 is the identifier in the second Chord ring. For example, node 1 on the first Chord ring has identifier 3 on the second Chord ring. Through these identifier pairs, two virtual Chord rings are organized. On the other hand, each data item has two replicas which are distributed in these two logical Chord rings. For example, if a data item has key 5, then this data item is stored in nodes with identifier pair $(7, 5)$ and $(5, 2)$, which means that if any identifier of a node's k identifiers matches a data item's key, this node will have a replica of that data item.

Since each data item has k replicas that are distributed in k Chord rings, a lookup request will try to find the numerically closest one to satisfy the query. At each hop during a lookup, the local routing information is used to select the current closest replica to route the request. Thus, a search may switch from one Chord to another to speedup routing by choosing the closest replica of the desired data item at each step.

1) Permutation of k Name Spaces: In overlaying k Chord rings, one of the fundamental problems is how to choose the k identifiers of each node. Assuming the size of identifier name space is N, the nodes' identifier name space can be expressed as $\mathbb{R} = (r_0, r_1, \ldots, r_{N-1})$, and we consider the name space of the first Chord ring to be $\mathbb{R}^0 = (0, 1, 2, \ldots, N-1)$. Thus, we can view all the possible nodes' identifiers on the other Chord rings as a permutation of $\mathbb{R}^0$. A well-chosen permutation which could make routing more efficient is desirable. In the following, we describe four simple and practical permutations for naming functionality.

Reverse Permutation Assuming the name space of the first Chord is $\mathbb{R}^0 = (0, 1, 2, \ldots, N-1)$, the permutation for the second Chord ring can be the reverse of $\mathbb{R}^0$, i.e., $(N-1, N-2, \ldots, 1, 0)$. Thus, for any node identifier Id_1 in the first Chord ring, we can obtain the identifier in the second Chord ring $Id_2 = N - 1 - Id_1$. This permutation, unfortunately, limits the number of Chord rings to just two.

Shift Permutation Suppose there are k Chord rings and a node's name is v, we can obtain its identifier in the first Chord ring by hashing its name: $P = hash(v)$. Based on this identifier, we can derive other $k-1$ identifiers of this node by adding a constant. Thus, from the entire name space perspective, other Chord rings can be viewed as simply Chord rings derived by shifting the original Chord ring. Let the name space of the first Chord be $\mathbb{R}^0 = (0, 1, 2, \ldots, N-1)$. The shift permutation is a cyclic shift of the name space $\mathbb{R}^i = (i, i+1, \ldots, N-1, 0, 1, \ldots, i-1)$. We can construct k-Chord system based on k equidistance shift permutations $\mathbb{R}^0, \mathbb{R}^{\frac{N}{k}}, \mathbb{R}^{\frac{2N}{k}}, \ldots$ etc. Thus, the identifiers of node r_p in the k-Chord system will be $(P, P + \frac{N}{k}, P + 2 \times \frac{N}{k}, \ldots, P + (k-1) \times \frac{N}{k})$ in the k Chord rings.

Random Permutation Another possible approach for choosing the k Chord rings is by random permutation. Based on the first name space $\mathbb{R}^0$, we can rearrange the sequence randomly to generate a new permutation for another Chord ring. A full mapping table is needed to keep track of complete permutations for each Chord ring. Each node, in this case, can get its k identifiers by querying the mapping table. It is expensive to maintain the mapping table for each node. In practice, we can obtain the permutations by applying a minimal perfect hashing function to generate the k identifiers of a node recursively. A perfect hash function is a collision-free hash function that maps different keys to distinct integers. A minimal perfect hash function can map different keys to distinct integers and has the same number of possible integers as keys, which means that n keys will map to $0..n-1$ without any collision. If a node's name is v, we can get the k identifiers as $P_1 = hash(v), P_2 = hash(P_1), \ldots, P_k = hash(P_{k-1})$. In this way, the values of P_i will be different with high probability.

Modular Permutation A modular approach can be used to obtain permutations that can cover the name space. Assuming that N is a prime number and m is an arbitrary integer, the m-modular permutation is obtained by skipping m consecutive elements, i.e., $\mathbb{R}^0_m = (0, m, 2m, 3m, \ldots)$ mod N. Since m and N are coprime, it is guaranteed that $\mathbb{R}^i_m = (i, i+m, i+2m, i+3m, \ldots)$ mod N is a permutation of $\mathbb{R}^0$. To obtain k different Chord rings we can choose different values of m to form k different identifier sequence permutations, where $i = 0$.

2) Routing with Multiple Chords: Routing and locating a desired data item efficiently is an extremely important criteria in a P2P system. Chord can route to the destination node by decreasing the distance by half after each hop. In the k-Chord system, we leverage k virtual Chords to speedup the routing process. Each node sets up a finger table for each Chord ring like Chord, and maintains a k-dimensional finger table for efficient routing.

During a lookup, each intermediate node resolves the query and checks if the destination is located within the range of its position and its successor on each Chord ring. If the destination is located within one of those ranges, it finds the desired node and just jumps to that node directly; if not, it

applies a greedy strategy to forward the query. The greedy strategy scans the k-dimensional finger table, finds the k predecessors of the destination on k Chord rings, and then chooses the node that is numerically closest to the destination as the next hop node. Thus, the lookup jumps can switch between the k Chords to speedup finding the closest replica of the desired data.

We will see later that the approach of overlaying k topologies to improve performance is case sensitive. If the interval between each hop is the same and long enough, it can work efficiently. However, the routing algorithm of Chord guarantees that the distance between the target and the location of the current node will decrease by half after each hop; with the increasing number of jumps, the distance to the destination becomes very short, and it has little chance to switch between the k Chords. This means that our k-Chord model can contribute a lot in the first several hops by switching between different Chords but does not help much in the subsequent hops.

B. Multiple Successor Lists of Constant Length

We now describe how to combine multiple chord rings with multiple successor lists. Chord[15] uses a successor list of variable length to increase system robustness, and CFS[6] uses it for data replication. In the hybrid system we propose, each node maintains a successor list of constant size d containing the node's first d successors. Therefore, the total number of successor lists is equal to the number of overlayed rings. The successor list contributes significantly for efficient routing, failure recovery and data replication.

1) Routing with Multiple Successor Lists: Since the Chord routing algorithm can cut the distance to the destination by half after each hop, the last several lookup hops may be within a very short distance to the destination. Our idea is to use the node's successor list as a part of the routing table. In this way, we might be able to avoid the last few hops by jumping directly to the destination when it is within the range of the successor list.

During a lookup, each intermediate node checks if the destination is located within the range of its position and the last node's position in its successor list. If the destination is located within that range, the node scans the successor list to find the successor corresponding to the searched key and jumps to that node directly; if not, it scans the finger table to find the closest predecessor of the searched key and jumps to that node to continue the query.

A key problem for this model is to choose a suitable length of a node's successor list that can cover a reasonable clockwise distance from it. Chord maintains r successors for failure recovery, and considers that $r = \Omega(\log n)$ can offer good performance. We, on the other hand, choose a constant value for the length of the successor list. We will show that such a choice guarantees good routing performance.

C. Data Lookup & Routing in the Hybrid System

In the hybrid system, each node maintains a k dimensional finger table and a successor list of size d. During a lookup, each intermediate node resolves the query and checks if the destination is located within the range of its position and its last successor in the successor list on each Chord ring. If the destination is located within one of those ranges, it finds the desired node and jumps directly to that node; if not, it applies the greedy strategy to forward the query. The greedy algorithm scans the k-dimensional finger table, finds the k predecessors of the destination on the k Chord rings, and then chooses the node that is numerically closest to the destination as the next hop node.

The k-Chord model reduces the distance between the source and the destination node sharply within the first few hops, which also helps compressing the node density between the current location and the destination. In other words, the nodes located within the distance of the last few hops have shorter intervals between each other, because if the distance is long enough, the lookup of the k-Chord model may try to switch to another Chord ring that has stored a replica much closer with high probability.

Although the k-Chord model may not locate the destination accurately within the remaining distance through small hops, immediate successors as a part of the routing table solves this problem by offering only one hop to locate the destination directly if the destination is located within the range of successors. The hybrid system achieves better performance than multiple Chords or multiple successor lists alone. It is also helpful for fault tolerance and re-routing in the event of failures.

D. Scalability

Since each node in the hybrid model needs to maintain a k-dimensional finger table and a successor list of size d on each Chord ring, the size of the routing table is $O(k(\log n + d))$, and the joining/leaving cost is $O(k(\log^2 n + d))$. With k and d being constant (e.g., $k = 4$ and $d = 20$), the size of the routing table is $O(\log n)$ and the total joining/leaving cost is $O(\log^2 n)$, the same as Chord.

When a new node joins the network, it joins the k chord rings. The newly joined node constructs the connection to its predecessor and successor on each Chord ring, and creates the finger table and the successor list for each Chord ring. Its successor on each Chord ring sends back the data associated with the keys that belong to the new node. When a node leaves, it transfers the data to corresponding successors on each Chord ring before it departs; it also notifies its predecessor and successor on each Chord ring to adjust their pointers. The successor lists of the related nodes are refreshed periodically.

E. Fault Tolerance

1) Data Replication: In the hybrid model, replicas of a data item are stored in the same location on k different Chords associated with its key and the d successors. The priority of lookup for a data item is routed first to the numerical closest destination, if the target node failed, it re-routes to that node's successor directly. If all d successor nodes fail, it abandons this Chord ring, and re-routes to the numerical closest Chords within the left valid Chords and does the same lookup mechanism until it finds the desired data or all the nodes storing the replicas of the data fail.

The successor list mechanism helps data replication and places replicas in a way that nodes can easily find them. We adopt the same replication scheme as CFS[6] in our model. The replicas of a data item are stored on r immediate successor nodes of the target node that is responsible for the keys associated with the data to increase data availability. Naturally, the number of replicas is smaller than the length of the successor list $r \leq d$. The target node k track of its r successors and propagates data to new replicas automatically when it detects that successors come and go.

Since nodes' identifiers are generated by hashing their IP addresses, the nodes close to each other on the logical Chord ring are not likely to be geographically close to each other. The failures are independent and failed nodes are distributed randomly on the Chord ring even if there are full failures of the nodes in a particular geographical region, or all the nodes that use a particular access link, or all the nodes that have a certain IP address prefix.

Actually, the successor list replication scheme is robust for both accidental failures and malicious failures based on the preceding reasons, since an adversary may be able to make some set of nodes fail, but has no control over the choice of the logical Chord ring.

2) Failures: From the lookup point of view, failures fall into two categories: random failures and target failures. Random failures occur accidently and affect the routing procedure on the intermediate nodes along the lookup path; target failures occur only on the nodes that store the desired data, which will affect the routing at the end of the lookup path. We use different fault tolerance strategies in each case.

Random Failures: When a massive random failure occurs, a lookup may encounter failed nodes along the lookup path. The fault tolerance strategy in this case is to bypass the failed nodes and continue along to reach the destination.

During each lookup iteration in Chord, every node selects, from its finger table, the largest *alive* node that precedes the target key to perform the next jump, and continues with routing. In the k-Chord model, we have adjusted our greedy algorithm to return the numerically closest *alive* node to the destination during each hop. The algorithm scans the k-dimensional finger table, finds the k valid closest predecessors of the destination on k Chord rings, and chooses the node that is numerically closest to the destination as the next hop node. Obviously, random failures may extend the mean lookup path.

Although the system can apply the successor list to refresh the routing table when failures happen, it will take some time to detect and correct it. Before the remaining nodes react to the failures, routing is performed in the same way as in Chord: every node chooses the largest *alive* node that precedes the target key from its finger table as the next jump node for the further lookup.

Target Failures: In the k-Chord system, there are k replicas for each data item. When the closest target node that contains the desired data item fails, the system will redirect the query message to another available node that has the desired data item.

One reasonable idea for re-routing is that, when reaching a failed target node, the query bypasses the failed node and jumps to another location from the current position of the same Chord based on its k-dimensional finger table, and continues with the lookup. However, it could confront a potential problem that it may re-route back to the failed node or jump between failed nodes indefinitely if there are more than one failed target nodes.

A valid but not efficient re-routing approach is that, when reaching a failed node that contains the desired data replica in one Chord, it will continue to look for the second closest location ignoring the Chord that contains the failed node, which is now considered an invalid Chord. The routing message will contain the information on previously found failed nodes, and the lookup hops will only switch between the valid Chords. The obvious disadvantage of this scheme is that, as the number of target node failures increases, the number of available Chords decreases along with the lookup performance.

Since replicas of a data item are stored at r ($r \leq d$) nodes succeeding its key, when the node that stores this data item fails, the lookup message can jump to its immediate successor to look for the desired data.

3) Failure Detection & Recovery: The successor list of a node can be used to detect and recover from failures automatically. Those successor pointers are then used to verify and correct finger table entries, which allows these lookups to be fast as well as correct.

Each node checks every entry in its finger table periodically; if there is a failure, it tries to find the failed node's alive successor to substitute it. As time passes, the scheme will correct the finger table entries and the successor list entries pointing to the failed node.

We can compare the hybrid model to Chord for the performance of lookup when failures happen. Since all the replicas of a data are distributed randomly from the view of the geographic network, Chord and the hybrid system should have the same data availability. However, because of the distinct data replication and routing algorithm on the overlay network, there is a little difference in their performances. Considering the better lookup performance and the very low probability of full linear failures of the hybrid system, the total performance of the hybrid system is better than Chord system.

III. Simulation on Hybrid-Chord

In our experiments, the size of the circle name space is $N = 10^6$. The number of nodes n is varied from 100 to 18,000. The n nodes are hashed by their randomly generated IDs and distributed uniformly along the Chord ring. In each simulation run, we choose 200 pairs of valid source nodes and desired keys randomly. The simulation is repeated 100 times in each case to get the average value for the length of the lookup path (in hops).

A. Lookup Performance

1) Effect of Different Permutations & Multiple Chord Rings: We now show the lookup performance of the k-Chord model alone for all four permutation schemes mentioned earlier.

Reverse Permutation The improvement (in the range 13 - 27%) in lookup performance for $k = 2$ is significant.

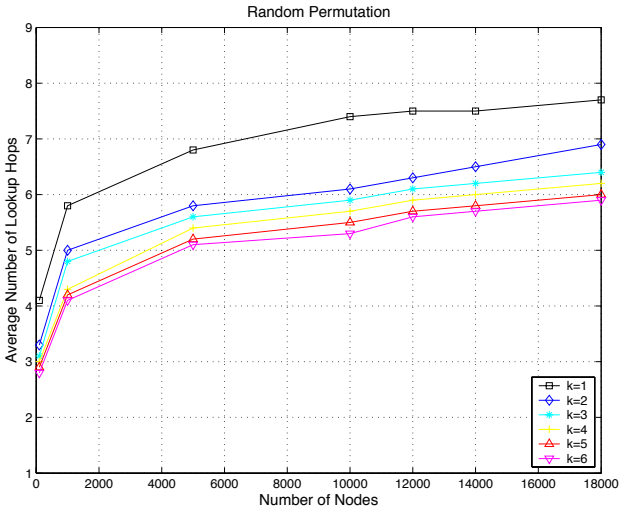

Fig. 2. Lookup Hops for Random Permutation

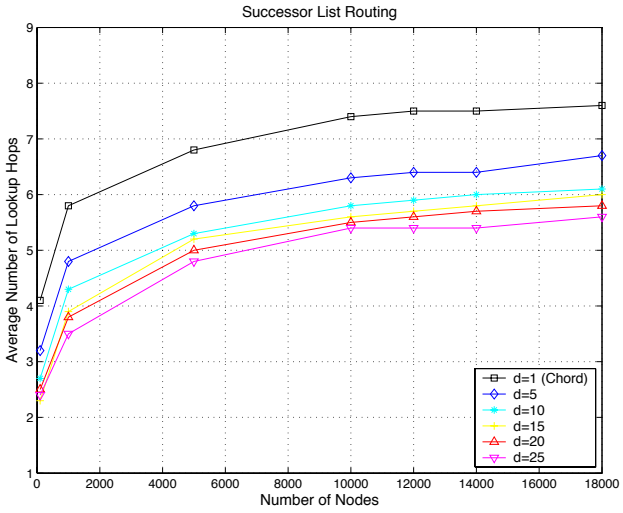

Fig. 3. Lookup Hops in Successor List of Different Sizes

TABLE I

EFFECT OF DIFFERENT m FOR MODULAR PERMUTATION, $k = 2$

m	$n = 1000$	$n = 5000$	$n = 10000$	$n = 15000$
5 (≈ 0)	4.96	5.92	6.15	6.44
17 ($\approx \log N$)	4.97	5.90	6.32	6.57
316 ($\approx \sqrt{N}$)	4.97	5.87	6.08	6.54
25000 ($\approx \frac{N}{4}$)	4.90	5.79	6.18	6.62
50000 ($\approx \frac{N}{2}$)	5.01	5.80	6.24	6.65
75000 ($\approx \frac{3N}{4}$)	4.92	5.82	6.29	6.49
99000 ($\approx N$)	4.96	5.81	6.23	6.61

Shift Permutation We get a significant performance improvement for $k = 2$, with the overall improvement in the range 18 - 27%. When $k \geq 5$, the improvement is insignificant. Note that all the jump switches happen only in the first hop of each lookup. This approach can be viewed as k replicas of each data item distributed uniformly on one Chord ring.

Random Permutation The performance of the random permutation (see Figure 2) is better than that of the shift permutation. A performance improvement between 23 - 32% is achievable depending on the value of k. As we can observe, most switches happen within the first three hops for each lookup. When $k = 2$, the average improvement is about 1.1 to 1.4 hops compared to Chord($k = 1$) for each lookup. When $k > 2$, the average improvement is only about 0.5 hop with increase in k.

Modular Permutation For this permutation, we chose a big prime number which is close to 10^5 as the size of the name space. Here, N is 100003, and different values for m are chosen from the range $(0, 100000)$. To see what is the best choice for m, we chose different values for m as: $m \approx (0, \log N, \sqrt{N}, \frac{N}{4}, \frac{N}{2}, \frac{3N}{4}, N)$ and varied n from 1000 to 15000 (see Table I).

In this particular case, we chose $k = 2$ because there is a substantial improvement when $k = 2$ in comparison with Chord (i.e. $k = 1$). In fact, the average improvement is about 0.8 to 1.3 drop in hops for $k = 2$. However, the drop becomes less and less significant as k is increased. The improvement is still good up to $k = 4$. It is very interesting to notice that the choice of m does not have a significant impact on the performance of data lookup in the case of modular permutation. Since different m gives similar performance, we randomly chose m for the experiments. We can see the performance improvement between 22 - 32% depending on k and the total number of nodes n.

In summary, the experiments show that the reverse permutation has the best performance for $k = 2$; the random permutation scheme offers a good routing performance, and the modular permutation scheme can accomplish similar performances.

The experiments on the four permutation schemes indicate that the lookup path length decreases with the increase in the number of overlayed Chord rings. However, the overhead associated with maintaining k rings also increases with k. Therefore, we need to find a trade-off between the performance improvement and the increasing overhead. The results (Figure 2) show that when k varies from 1 to 2, the lookup performance improvement is very much significant. After that the improvement slows down. When $k \geq 5$, the improvement becomes insignificant. Thus, $k = 2, 3$ or 4 can be a very good choice. If nodes join and leave the network frequently, the performance improvement for $k = 2$ will certainly outweigh any extra overhead; however, if the network does not change significantly, $k = 4$ appears to be the best choice.

2) Effect of Length of the Successor List: We now show the lookup performance of the successor list model alone varying the length of the list. In the experiment, the length of the successor list (d) is varied from 1 to 25. The n nodes are hashed by their random generated IDs and distributed uniformly on the Chord ring. We can see from Figure 3 that we can achieve a significant improvement in the lookup performance for d up to 10. When $d > 20$, the performance improvement becomes considerably insignificant. Overall, an improvement in the range of 26 - 40% is possible depending on d and n.

The routing performance improves as we increase d, but the overhead also increases at the same time. We, therefore, need to find an optimal value of d. When $d > 20$, there is not much improvement because of the overhead which seems to

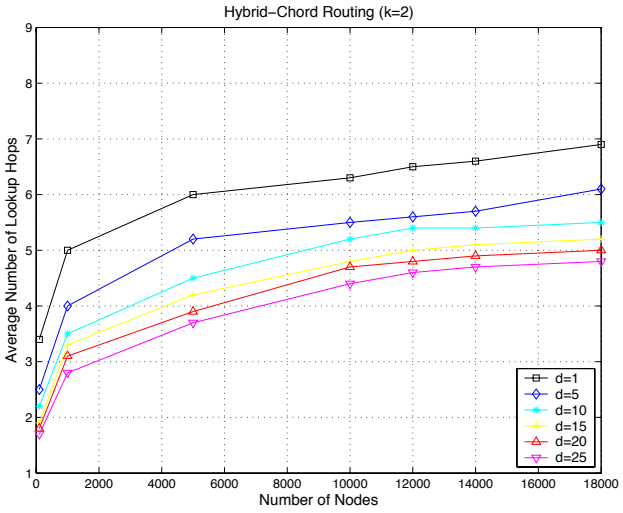

Fig. 4. Lookup Hops for the Hybrid System for $k = 2$

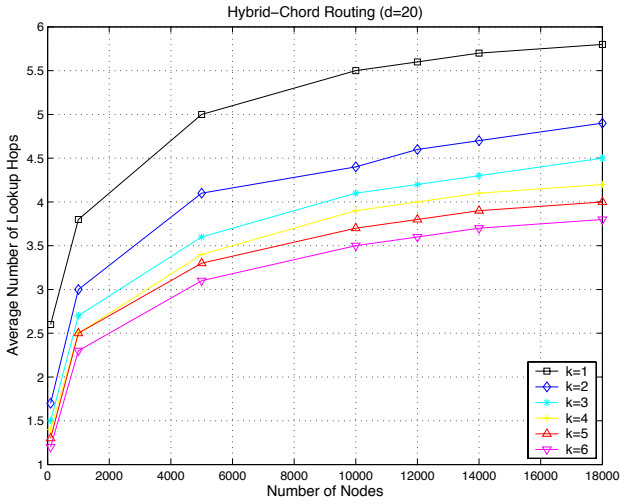

Fig. 5. Lookup Hops of the Hybrid System for $d = 20$

indicate that $d = 20$ is a good choice. In the following, we give an intuition of why $d = 20$ is a good choice in our setting. Assume that the size of the ring name space is 100,000, and the number of nodes is 10,000. Through hashing function, all 10,000 nodes can be distributed uniformly on the ring, which means the interval of two neighboring nodes is about 10. Thus, $10 \sim 15$ successors of a node can almost cover the distance of about $100 \sim 150$ possible nodes close to it. The distances of the node's finger nodes are about $2^0, 2^1, ..., 2^i, ..., 2^{\log N}$; when $i = 7$, the distance is 128 (when $i = 8$, the distance of $2^8 = 256$ will require at least 26 successors in the ideal case), and $i < 4$ will be covered by the first successor. So 4 (=7-3) hops distance may be covered by the successor list with the length of $10 \sim 15$. However, in real system, each hop can at least reduce the distance to the destination by half, so the simulated result is that, in most of the lookups, the last $2 \sim 5$ hops can be merged into one hop by successor list routing. Under this model, assuming the length of the successor list $d = 20$, we expect to see $1 \sim 2$ drop in the average lookup hop.

3) Lookup Performance of Hybrid-Chord: We now consider the hybrid system. From the lookup performance perspective, we study:

- the effect of the length of the successor list on the lookup performance for a fixed number of overlayed Chord rings. The number of Chord rings was fixed at $k = 2$, and the successor list length d was varied from 1 to 25 in our experiment,
- the effect of the number of Chord rings on the lookup performance for the successor list of an ideal length. The successor list length was fixed at $d = 20$, and the number of Chord rings k was varied from 2 to 7 with random permutation used in naming.

With $k = 2$, we obtain a lookup performance improvement (Figure 4) in the range 30 - 50%. Similarly for $d = 20$, the improvement (Figure 5) is in the range 38 - 53%, which is very significant. Therefore, $k = 2, d = 20$ seems to be a good combination for the hybrid system.

TABLE II
EFFECT OF DISTRIBUTION DENSITY ON THE HYBRID SYSTEM ($d = 20$)

N	$k = 1$	$k = 2$	$k = 3$	$k = 4$
20000	5.26	4.34	4.06	3.87
50000	5.31	4.33	4.03	3.99
100000	5.38	4.49	4.11	3.91
1000000	5.41	4.58	4.25	4.01
10000000	5.48	4.70	4.31	4.10

B. Effect of Distribution Density

In Chord, the nodes are distributed uniformly along the Chord ring; hence, the lookup performance depends only on the number of nodes in the network, which means that if n is fixed, the node density has little effect on the routing performance. We will see that this is also true for the hybrid system.

In our simulation, the number of nodes is fixed (i.e., $n = 10^4$), the size of the name space N is varied from 20000 to 10^7, the number of Chord rings k is varied from 1 to 4 and the length of the successor list is fixed (i.e., $d = 20$) as shown in Table II. The experiment indicates that the lookup path length increases very little with N for each value of k. We, therefore, conclude that under uniform node distribution, the routing cost of the hybrid system depends mostly on the number of nodes (n) in the system and not on the size of the name space (N).

C. Effect of Simultaneous Node Failures

After a node in the hybrid system fails, some time will pass before the remaining nodes react to the failure, by correcting their finger tables and successor pointers and by copying replicas to maintain the replication. The hybrid system is able to perform lookups correctly and efficiently before this recovery process starts, even in the event of massive failure.

To test that, 1000 data items were inserted into a 1000-node system, each data item having 6 replicas. For $k = 2$, $d = 20$, a fraction p (varied from 10% to 50%) of all nodes were randomly chosen as failure nodes. After that, we performed 10000 random lookups. For each lookup, we recorded if the lookup was a success and if it was, we calculated the

TABLE III
HYBRID AND CHORD LOOKUP FAILURE RATE

p(fraction of failure)	20%	30%	40%	50%
Chord $(k = 1, r = 6, d = 20)$	0	0.002	0.010	0.016
Hybrid $(k = 2, r = 3, d = 20)$	0	0	0.009	0.014

TABLE IV
HYBRID AND CHORD LOOKUP PATH LENGTH FOR FAILURES

p(fraction of failure)	10%	20%	30%	40%	50%
Chord $(k = 1, r = 6, d = 20)$	6.0	6.2	6.5	6.8	7.2
Hybrid $(k = 2, r = 3, d = 20)$	3.5	4.1	4.8	5.6	6.6

lookup path length. We then derived statistics of the lookup success rate and the average lookup path length (only for the successful lookups).

Table III shows the lookup failed rate when the failure fraction p varies from 10% to 50%, and the number of the successors r that store the data replicas. The size of the successor list is $d = 20$ in both systems. The result shows that our hybrid system can always find the desired data with high probability, and has a similar data availability as Chord. Table IV shows the average lookup path length when failures occur. The result indicates that the our hybrid system has better lookup performance when failures occur. Based on the above observations, we can claim that our hybrid system outperforms the Chord in handling node failures.

Chord considers that if the successor list has length $d = \Omega(\log n)$, both the success rate and the performance of Chord lookups will not be affected even by massive simultaneous failures. Furthermore, it has been shown that, if the successor list of length $d = \Omega(\log n)$ and every node fails with independent probability $1/2$, the system can find the closest living desired node and the expected lookup time is $O(\log n)$. However, with the evolution of the network, a node cannot know the exact number of nodes existing in the network at a certain time. More practically, in our model we use a reasonable constant number ($d = 20$) as the length of the successor list. Assuming the independent failure probability of a node is $1/2$, the full failure for a successor list is $(\frac{1}{2})^{20}$ which is very small. It means that the data items are always available with high probability.

The correctness of lookup scheme relies on the fact that each node knows its successor. Failure nodes will result in incorrect successor pointers, and incorrect successor will lead to incorrect lookup. To increase robustness, in the same way as Chord, each node maintains a successor list of size d, containing the node's first d successors. If a node's immediate successor does not respond, the node can substitute the second entry in its successor list. All d successors would have to simultaneously fail in order to disrupt the Chord ring, an event that can be made very improbable with modest values of d. Assuming each node fails independently with probability p, the probability that all d successors fail simultaneously only p^d. Increasing the size d of successor list can strengthen system robustness.

IV. REDUNDANT D2B

One of the latest trends in P2P system design focuses on maintaining constant degrees to achieve efficient routing performance. Some latest distributed hash table based P2P systems ([9], [8], [11], [1]) use the *De Bruijn* graph to construct P2P overlay networks with constant degrees and can achieve $O(\log n)$ routing performance. In this section, we propose a variation of the D2B system [8], which improves the lookup performance and data availability while maintaining a small degree.

A. Overview of De Bruijn Graph

We first briefly describe the De Bruijn Graph, which is the basis of D2B, and of our variation.

A *De Bruijn* graph $B(d, b)$ is a directed graph whose nodes are all strings of length b on the alphabet $\{0, \ldots, d-1\}$, and there is an edge form any node $x_1x_2 \ldots x_k$ to the d nodes $x_2 \ldots x_k\alpha$, for $\alpha = 0, \ldots, d-1$. $B(d, b)$ has d^b nodes, in-degree and out-degree d, and diameter b. In the following, we will consider boolean alphabets, that is $d = 2$. Routing from $x_1 \ldots x_b$ to $y_1 \ldots y_b$ is achieved by following the route $x_1 \ldots x_b \rightarrow x_2 \ldots x_b y_1 \rightarrow x_3 \ldots x_b y_1 y_2 \rightarrow \ldots \rightarrow x_b y_1 \ldots y_{b-1} \rightarrow y_1 \ldots y_b$. For example, if it routes from 000 to 111, the route is $000 \rightarrow 001 \rightarrow 011 \rightarrow 111$. A shorter route is obtained by looking for the longest sequence that is suffix of $x_1 \ldots x_b$, and prefix of $y_1 \ldots y_b$. If there is such a sequence $x_i \ldots x_b = y_1 \ldots y_{b-i+1}$, then the shortest path from $x_1 \ldots x_b$ to $y_1 \ldots y_b$ is $x_1 \ldots x_b \rightarrow x_2 \ldots x_b y_{b-i+2} \rightarrow x_3 \ldots x_b y_{b-i+2} y_{b-i+3} \rightarrow \ldots \rightarrow x_{i-1} \ldots x_b y_{b-i+2} \ldots y_{b-1} \rightarrow y_1 \ldots y_b$. For example, if we want to route from 111000 to 000111, the path is $111000 \rightarrow 110001 \rightarrow 100011 \rightarrow 000111$.

Routing is very simple in a complete De Bruijn graph. The D2B system aims to maintain a dynamic De Brujin graph to create a P2P topology where routing is still simple and efficient.

B. Redundant D2B

In D2B, there is no fault tolerance. For a specified start node and a key, the lookup path is unique. If a node in the lookup path fails, that lookup process is unavailable, furthermore, all lookup paths through that node are unavailable. In this section, we propose a variation that incorporates fault tolerance while improving the lookup performance.

Let us call $i-$*successors* of a node x, the set of nodes that x can reach in i hops or less. Clearly in a full De Bruijn, each node has two 1-successors and 2^i i-successors.

Similar to Hybrid-Chord, we maintain a successor list for each node in the D2B system to improve the routing performance and data availability. Since the number of successors is exponentially increasing and we want to keep the degree of the system small, we propose to maintain a list of successors of constant length. In this paper, we have studied the cases $i = 2$ and $i = 3$ and have observed substantial improvements in routing, data replication, and fault tolerance. Node joining and leaving is managed exactly in the same way as in D2B. Routing proceeds in a similar way, exploiting the additional connections.

Successor Routing. In our variation, routing proceeds in a similar way as in the D2B system, where the additional links are used as "shortcuts". In other words, for each hop a

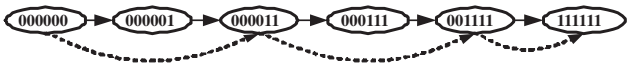

Fig. 6. The Successor Routing in D2B

message is forwarded in a greedy way to the node closest to destination. In doing so, a message can "skip" $k-1$ nodes in its original routing path if k successors are kept (an exception is the last hop, which may allow to skip a variable number of nodes between 0 and $k-1$, depending on the location of the target node). If the D2B routing performance is P, then the improved D2B can achieve $\frac{1}{k}P$ routing performance.

Figure 6 shows the sample routing for the improved D2B system, $k = 2$, which means each node will maintain a set of 2-successors; the number of out-going link for each node is then $2^1 + 2^2 = 6$. Assume there is a lookup initiated from the node with label "000000", and the destination node is "111111...". The original routing would forward the request in 5 hops. In successor routing, each node is connected also to the nodes that it can reach at distance 2, and can forward the request to the successor closest to the destination. Thus, compared to the original routing, one node will be skipped at each hop, which means close to half of the nodes in the original lookup path are skipped in total, and the final number of lookup hops is 3.

Since it appears that small successors' distances can significantly improve the performance, and since increasing the distance of successors leads to exponential increase in the number of successors (and thus of the network's degree), we have decided to perform experiments with 2-successors and 3-successors only. As we will see, the results are quite good; in fact, we have improvements in performance maintaining a small degree.

Data Replication. The successor list mechanism also helps data replication and places replicas in such a way that nodes can easily find them. The replicas of a data item are stored on $r, r \leq k$ level immediate successor nodes of the target node that is responsible for the keys associated with the data to increase data availability. The target node k track of its r successors and propagates data to new replicas automatically when it detects that the successors come and go.

Because nodes' identifiers are generated by hashing their IP addresses, the nodes close to each other on the logical De Bruijn graph are not likely to be geographically close to each other. The failures are independent and failed nodes are distributed randomly on the network even if there are full failures of the nodes in a particular geographical region, or all the nodes that use a particular access link, or all the nodes that have a certain IP address prefix. Thus, the system has high data availability with high probability.

System Recovery. The successors of a node can be used to detect and recover from failures automatically. Each node will periodically send messages to check if any of its neighbors (all successors and predecessors) fails, and try to recovery the failure.

Fault Tolerance. Consider first the case of random failures. Although the system can apply the successors to refresh the routing table when failures happen, it will take some time to detect and correct it. Before the remaining nodes react to the failures, routing is performed in the same way, except trying to bypass the failed node. In the case of target failures, since replicas of a data item are stored at the nodes succeeding its key, it can jump to its immediate successor to look for the desired data efficiently when the node that stores this data fails. If all the direct successors fails, it continues with the next successor until the data is found or the list of successor is exhausted.

TABLE V
ROUTING PERFORMANCE FOR D2B, K=1

n	$\log n$	Max	Min	Average	Chord
1000	10	12	1	8.2	5.2
2000	11	12	1	9.3	5.8
5000	12.3	18	1	10.6	6.7
10000	13.3	15	4	11.6	7.2
15000	13.8	14	1	12.1	7.5
20000	14.3	16	1	12.6	7.7

TABLE VI
ROUTING PERFORMANCE FOR REDUNDANT D2B, K=2

n	Max	Min	Average	D2B	Successors
1000	6	0	4.4	8.2	6.5
2000	6	1	4.8	9.3	6.4
5000	7	1	5.4	10.6	6.8
10000	8	1	6.01	11.6	6.7
15000	8	1	6.4	12.1	6.3
20000	8	1	6.58	12.6	6.5

V. SIMULATION FOR REDUNDANT D2B

A. Routing Performance of D2B

In the simulation, nodes join the D2B network one by one, until the number of nodes is n. Then, the system is considered steady. We randomly choose 1000 pairs of the source node and m-bits target key string and simulate the lookup experiment. We then compute the average hops for each lookup. The results in Table V indicate that the routing performance of D2B is worse than that of the Chord. The average number of lookup hops is smaller than $\log n$, but greater than that in the case of Chord. The average number of lookup hops stabilizes after about 10 iterations of testing.

B. Routing Performance of Redundant D2B

The simulation conditions are the same as in D2B. The results for $k = 2$ and $k = 3$ are shown in Tables VI and VII. Since D2B is not a perfect De Bruijn topology, the number of each node's out-degree is variable and depends on the dynamics of insertions and removals. The tables also show the average number of out neighbors of each node.

As expected, the results suggest that the number of lookup hops is in the order of $\frac{1}{k}\log n$. The experiments also show that the average number of successors is about $\sum_{i=1}^{k} 2^i$ (for example, for $k = 3$, the number of the successors is about 16). Compared to Chord, we can observe that the the *Redundant D2B* has better performance. Note that the number of successors is independent of the size of the network n.

TABLE VII
ROUTING PERFORMANCE FOR REDUNDANT D2B, K=3

n	Max	Min	Average	D2B	Successors
1000	5	1	3.0	8.2	16.1
2000	5	0	3.45	9.3	15.7
5000	5	1	3.85	10.6	16.1
10000	5	2	4.3	11.6	16.3
15000	6	1	4.42	12.1	16
20000	6	1	4.54	12.6	15.9

TABLE VIII
LABEL LENGTH OF A NODE

n	$\log n$	Longest	Shortest	Average
1000	10	13	8	10.25
2000	11	14	9	11.2
5000	12.3	16	9	12.6
10000	13.3	17	11	13.6
15000	13.87	17	11	14.1
20000	14.3	18	12	14.5

TABLE IX
AVERAGE DEGREES OF A NODE

n	out	in	max out	min in
1000	2.12	2.12	12	6
2000	2.11	2.11	13	7
5000	2.13	2.13	20	10
10000	2.12	2.12	19	9
15000	2.12	2.12	17	9
20000	2.12	2.12	16	11

TABLE X
AVERAGE LOOKUP HOPS AND ROUTING TABLE SIZE FOR DIFFERENT SYSTEMS

n	Chord		Hybrid Chord		D2B		Redundant D2B	
	hops	size	hops	size	hops	size	hops	size
1000	5.2	10	2.5	60	8.2	2.12	3.0	16.1
2000	5.8	11	3.1	64	9.3	2.11	3.45	15.7
5000	6.7	13	3.4	69	10.6	2.13	3.85	16.1
10000	7.2	14	3.9	73	11.6	2.12	4.3	16
15000	7.5	14	4.1	75	12.1	2.12	4.42	16.3
20000	7.7	15	4.3	77	12.6	2.12	4.54	15.9

C. Observations about D2B

In this section, we make some observations that apply to both D2B and our variation.

Balance. A balanced network would guarantee that the keys/data are distributed uniformly by the key distribution rules of D2B. An interesting question is whether a dynamic De Bruijn network, as described in D2B, is balanced after joining n nodes. We consider the network balanced if the size of the labels of each node is close to $\log n$ (which would be the length of the node's label in a complete De Bruijn). In the experiment, we have randomly generated a 32-bits temporary label for a node's joining and we have reported the results in Table VIII. It can be seen that the average length of node labels is very close to $\log n$. However, the difference between the longest and shortest length is not small.

Constant Degree. Another interesting experiment is to observe the behavior of the degree for the dynamic De Bruijn of D2B. In the complete De Bruijn, each node has two parents (in-degree) and two children (out-degree). However, in the incomplete D2B, one node may have a variable number of children. Thus, we want to know the average degrees for the nodes for a certain n, where n is the number of nodes in the network. We need to verify if the size of each node's routing table is still constant. Table IX shows that, although the maximum number of in-degree and out-degree is larger than 2, the average in-degree and out-degree is very close to 2, and this value is independent of the size of the network. Thus, we can conclude that the size of routing table is constant.

VI. CONCLUDING REMARKS

Distributed hash tables are proving to be a useful foundation for more complex distributed applications. They provide a robust and scalable approach to the problem of content distribution and can be used to offer fault tolerance capabilities to P2P applications. Most recent research on P2P have focused on achieving efficient routing performance with constant degrees.

Our paper presents redundant P2P systems to improve the routing performance and data availability. In particular, we have studied a variation of Chord (*Hybrid-Chord*) and a variation of D2B (*Redundant D2B*). Both variations use the idea of maintaining a successor list to improve the lookup performances, as well as fault tolerance and availability. From our experiments, it emerges that using a small extra storage to keep the routing information about the successors allows us to obtain substantial improvements. The following table summarizes our results comparing them with the Original Chord and D2B, where Hybrid Chord with $k = 4$ and $d = 20$, and Redundant D2B with $k = 3$.

Related Work. Several variations of Chord have been proposed in the literature. In F-Chord[4], for example, the distances between the chords follows the Fibonacci sequence; in this way the authors show that the average number of hops for lookups decreases while the size of the routing table remains the same. In [5] similar improvements are obtained by adding an extra *random* chord to each node. Another variant of Chord is described in [2] where an overlay topology with multiple chord rings (called DKS) is introduced to improve routing performance. In DKS, the specified key is stored in a certain node. At the beginning of the search, the search space is equal to the whole identifier space. At each step of the search, the current search space is divided into k equal parts. Each part is under the responsibility of a well chosen node. This partitioning of the search space is repeated until the k equal parts contain an element each. This procedure can be viewed as searching on multiple Chord rings, and each ring owns $\frac{1}{k}$ part of the former ring. The partitioning of the search space leads to an improvement in lookup performances.

In the literature, there are several P2P systems based on the De Brujin graph (e.g., see Koorde [9] and D2B [8]). Koorde and D2B differ in the sense that Koorde uses the same key-space as Chord where a key $\kappa \in [0, 2^m - 1]$ induces connections to positions $2\kappa \bmod 2m$ and $2\kappa + 1 \bmod 2m$, whereas, in D2B, a node's label is a prefix of its managed keys.

REFERENCES

[1] I. Abraham, B. Awerbuch, Y. Azarand, Y. Bartal, D. Malkhi and E. Pavlov, "A generic scheme for building overlay networks in

adversarial scenarios," in *Proc. of the 17th International Parallel and Distributed Processing Symposium (IPDPS'03)*, 2003, pp. 40.2.

[2] L. Onana Alima, S. El-Ansary, P. Brand and S. Haridi, "Dks(N, k, f): A family of low communication, scalable and fault-tolerant infrastructures for p2p applications," in *Proc. 3rd Intl. workshop on Global and Peer-To-Peer Computing on Large Scale Distributed Systems(CCGRID'03)*, 2003, pp. 344–350.

[3] I. Clarke, O. Sandberg, B. Wiley and T. W. Hong, "Freenet: A distributed anonymous information storage and retrieval system," in *ICSI Workshop on Design Issues in Anonymity and Unobservability*, 2000, pp. 46–66.

[4] G. Cordasco, L. Gargano, M. Hammar, A. Negro, V. Scarano, "F-chord: improved uniform routing on chord," in *11th Int. Colloquium on Structural Information and Communication Complexity(SIROCC'04)*, 2004, pp. 89–98.

[5] G. Cordasco, L. Gargano, M. Hammar, A. Negro, V. Scarano, "Non-uniform deterministic routing on f-chord(α)," in *Int. Workshop on Hot Topics in Peer-to-Peer Systems(HOT-P2P'04)*, 2004, pp. 16–21.

[6] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris and I. Stoica, "Wide-area cooperative storage with CFS," in *Proc. 18th ACM Symposium on Operating Systems Principles(SOSP'01)*, 2001, pp. 202–215.

[7] M. Datar, "Butterflies and peer-to-peer networks," in *Proc. European Symposium on Algorithms(ESA)*, vol. 2461, 2002, pp. 310–321.

[8] P. Fraigniaud and P. Gauron, "D2b: a de bruijn based content-addressable network"," *Theoretical Computer Science, Issue on Complex Networks*, vol. 355, no. 1, pp. 65–79, 2006.

[9] F. Kaashoek and D. R. Karger, "Koorde: A simple degree-optimal hash table," in *Proc. 2nd International Workshop on Peer-to-Peer Systems(IPTPS'03)*, 2003, pp. 98–107.

[10] D. Malkhi, M. Naor and D. Ratajckzac, "Viceroy: A scalable and dynamic emulation of the butterfly," in *Proc. 21st ACM Symposium on Principles of Distributed Computing(PODC)*, 2002, pp. 183–192.

[11] M. Naor and U. Wieder, "Novel architectures for p2p applications: the continuous-discrete approach," in *Proc. 15th annual ACM Symposiumon Parallel Algorithms and Architectures*, 2003, pp. 50–59.

[12] S. Ratnasamy, P. Francis, M. Handley, R. Karp and S. henker, "A scalable content-addressable network," in *Proc. of ACM SIGCOMM*, 2001, pp. 161–172.

[13] M. Ripeanu, I. Foster and A. Iamnitchi, "Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design," in *IEEE Internet Computing Journal*, vol. 6, no. 1, pp. 50–57, 2002.

[14] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in *LNCS 2218*, 2001, pp. 329–350.

[15] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *ACM SIGCOMM*, 2001, pp. 149–160.

[16] U. Wieder and M. Naor, "A simple fault tolerant distributed hash table," in *2nd International Workshop on Peer-to-Peer Systems(IPTPS'03)*, 2003, pp. 88–97.

[17] M. Xie, "A decentralized redundant peer-to-peer system based on chord: Routing, scalability, robustness," Master's thesis, University of Ottawa, Ottawa, Canada, 2004.

[18] B. Y. Zhao, J. D. Kubiatowicz and A. D. Joseph, "Tapestry: An infrastructure for fault-tolerant wide-area location and routing," U. C. Berkeley, Tech. Rep. CSD-01-1141, 2000. 2003, pp. 88–97.

Paola Flocchini is a Professor at the School of Information Technology and Engineering (SITE) at the University of Ottawa, Canada, where she is also University Research Chair in distributed computing. She received her Ph.D. degree in computer science from the University of Milan, Italy, in 1995. Her research interests are in the areas of distributed computing, distributed algorithms, structural information, sense of direction, mobile computing, cellular automata, fuzzy cellular automata and discrete chaos.

Amiya Nayak (SM'04) received the B.Math. degree in computer science from the University of Waterloo, Waterloo, ON, Canada, and the Ph.D. degree in systems and computer engineering from Carleton University, Ottawa, ON, Canada, in 1982 and 1991, respectively. He is an Associate Professor at the School of Information Technology and Engineering (SITE), University of Ottawa, Canada. He has over 15 years of industrial experience in software engineering in avionics and telecommunication applications, working at CMC Electronics and Nortel Networks prior to joining the University of Ottawa in 2002. He has been an Adjunct Researcher Professor at Carleton University since 1994. His research interests are in the areas of fault-tolerant computing, ad hoc networks, and distributed computing.

Ming Xie received the B.Eng. degree in computer science and engineering from the Zhejiang University, China, and the M.Sc. degree in computer science from the University of Ottawa, Canada, in 1998 and 2004, respectively. He is a Ph.D. candidate at the School of Information Technology and Engineering (SITE), University of Ottawa, Canada. He has over 5 years of industrial experience in software development. His research interests are in the areas of distributed computing, mobile computing, and peer-to-peer systems.