Université d'Ottawa · University of Ottawa

Université d'Ottawa · University of Ottawa

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

Ming XIE

AUTEUR DE LA THÈSE - AUTHOR OF THESIS

Master of Computer Science

GRADE - DEGREE

School of Information Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT - FACULTY, SCHOOL, DEPARTMENT

TITRE DE LA THÈSE - TITLE OF THE THESIS

A Decentralized redundant Peer-to-Peer System Based on Chord : Routing, Scalability, Robustness

P. Flocchini

DIRECTEUR DE LA THÈSE - THESIS SUPERVISOR

CO-DIRECTEUR DE LA THÈSE - THESIS CO-SUPERVISOR

EXAMINATEURS DE LA THÈSE - THESIS EXAMINERS

A. Nayak

N. Santoro

J.-M. De Koninck, Ph.D.

LE DOYEN DE LA FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

DEAN OF THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

A Decentralized Redundant Peer-to-Peer System Based on Chord: Routing, Scalability, Robustness

by

Ming Xie

A thesis submitted to
the Faculty of Graduate and Postdoctoral Studies
in partial fulfilment of
the requirements for the degree of
Master of Computer Science

Supervisor: **Dr. Paola Flocchini**

Ottawa-Carleton Institute for Computer Science
School of Information Technology and Engineering
University of Ottawa
Ottawa, Ontario, Canada

Your file Votre référence
ISBN: 0-494-01648-5
Our file Notre référence
ISBN: 0-494-01648-5

Abstract

Peer-to-peer systems can share the computing resources and services by directly communicating within a widely distributed network. Our research focuses on locating, routing, scalability and robustness in the decentralized redundant P2P system.

Chord[1] is an efficient peer-to-peer lookup protocol for its simplicity, provable correctness and performance. It can locate a key through a sequence of $O(\log n)$ other nodes toward the destination, where n is the total number of nodes.

In this thesis, we present three enhanced models based on Chord to improve the routing performance and data availability. The k-Chord model helps lookup to approach the destination rapidly within the first several hops. The successor routing model applies the successor list as a part of the routing table to locate the destination accurately by merging the last several hops into one hop if the current lookup location is located within the range of the successors. The hybrid model is a combination of the former two models, and nearly halves the routing path length of Chord system. We also discuss the scalability and fault tolerance of the three models.

Acknowledgments

First and foremost, I would like to express my sincere gratitude and respect to Dr. Paola Flocchini, the supervisor of my Master thesis. Her invaluable and patient guidance, enthusiastic encourage and support accompany me in every stage of my thesis research. I feel extremely fortunate and honored to be her student.

I also thank Dr. Nicola Santoro, the team leader in our P2P research group, for his valuable suggestions and great support on my thesis work. I am grateful to Dr. Amiya Nayak and other research members in our group, Mr. Elio Velazquez, Mr. Vaughan Loucks and Mrs. MiaoJun Huang.

Finally, I want to express my deepest feeling to my family. My special thanks go to my parents, Mr. Houxin Xie and Mrs. Jifen Zhang. I especially thank my wife Ying Liu and my lovely daughter Qinghui Xie. Without their love and encouragement, I would not pursue new goals again and again.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 P2P Systems

Peer-to-peer computing describes the current trend toward utilizing the full resources available within a widely distributed network of nodes with increasing computational power. These resources include the exchange of information, processing cycles, cache storage, and disk storage for files.

A peer-to-peer system (P2P) is formed by a large number of nodes that can join and leave the system at any time and have equal capabilities without any centralized control or hierarchical organization. It can share the computing resources and services by directly communicating within a widely distributed network. Each node in the system may be a server, client or router.

1.2 Definitions

The fundamental features of P2P systems are:

- **Performance**: The total time in data read, insert and delete operations. Factors include the locality of data, load balancing, the efficiency of the locating algorithm,

and the efficiency of the routing protocol. In this thesis, we use hops to evaluate routing efficiency, and number of messages to evaluate node joining and leaving cost.

- **Topology**: The organization of nodes and data in a special logic network structure for efficient locating and routing, minimizing storage and enhancing the scalability of the system.

- **Naming**: The method used to represent shared data objects, network addresses of the nodes, and the structure of routing requests across the network. An appropriate addressing scheme can increase the performance.

- **Locating & Routing**: The algorithms used to efficiently locate data and rout to the node that stores the desired data. Efficient algorithms minimize the overhead of requests/queries and increase both scalability and performance.

- **Scalability**: The ability of the system to remain tractable with an increasing number of nodes and data. Nodes can join and leave the system freely, and the performance of the system should be only affected locally after a join or a leave.

- **Reliability**: The avoidance of failure within the system and the ease of recovery if a failure occurs and the availability of multiple paths to data. It includes data replication, node failure detection and recovery; the existence of multiple paths guarantees the location of information in case of a single point of failure.

The physical machines/peers in the network are called **node**s and the data items are called **object**s in the thesis. **Name space** refers to a set of names where all names are

unique. A **name** is used to represent a node and one node may have several names by different criteria, but generally, each node has only one name within one name space. A typical name for a node in Internet is its IP address. An **identifier** refers to the unique digital name of a node within a certain integer name space. In P2P systems, it can be obtained by hashing the name of a node with a suitable hash function. Since we use distinct integers to mark each node's name here, name and identifier have the same meaning in the thesis if there is no special claiming. A **key** is the unique identifier of a data item and can be gained from hashing the name of a data item.

A **Hash function** can map keys to integers, usually to get an even distribution on a smaller set of values. A **Hash table** is a dictionary in which keys are mapped to array positions by a hash function. Having more than one key map to the same position is called a collision.

Since P2P systems are dynamically changing, the routing tables of the nodes will have to be updated. In order to ensure that the routing scheme remains scalable in a dynamic environment, the hashing scheme, which is also called **consistent hashing**, must have the following properties:

- balance - the keys must be evenly distributed over nodes
- smoothness - when a node joins or leaves, the number of keys that are moved is the minimal number required to preserve the balance
- spread - a given key must be assigned to a small number of nodes (or, better, to a single node)

1.3 Motivation

Peer-to-peer systems can share the computing resources and services by communicating directly within a widely distributed network. They support file sharing, instant message, distributed computing, collaboration, etc. Peer-to-peer file sharing systems are now one of the most popular Internet applications and have become a major source of Internet traffic. Thus, it is extremely important that these systems are scalable and can efficiently locate the node that stores the desired data in a large system. 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.

In P2P system there is typically a large number of nodes that are willing to share objects (files, resources) available to them. They want to be able to search for such objects using location independent identifiers (keys). The challenge is to build a self-organizing, decentralized system that will allow every connected node to share its objects and to search for other objects. The P2P system is very dynamic: nodes may join or leave; nodes may add/remove objects on/from their computers. The system must have the following functionality:

- **Object query** for a key- if the object associated with the *key* exists in the system, then the query must successfully return it. Let the *source* node be the node that issues the query and the *destination* node the one that has the requested object.
- **Join(A,B)** - node A joins the network through node B that is already in the network
- **Leave(A)** - node A leaves the network

- **Publish(key, object)** - the owner node publishes an object, identified by its *key*
- **Unpublish(key,object)** - the object identified by *key* is no longer available at the owner node.

The nodes will exchange messages over the overlay network that is built on top of the physical network. The queries must be routed in the overlay network in order to reach a proper destination where the query can be answered. Once the location of the requested object is known, the transfer of the object between the two nodes will take place outside the overlay network.

Transferring the requested object between the nodes is naturally scalable. Therefore the scalability of the P2P system depends crucially on the scalability of the routing protocol. For a routing protocol to be scalable, the following objective functions must remain low for a large number of nodes:

- the query time, measured in the number of hops in the overlay network
- the number of messages exchanged for answering a query
- the amount of data exchanged to maintain the local state (the information about routing table, neighbor set etc.) at each node
- the amount of memory needed by each node to keep the local state

Obviously, if no local state is kept, the query must be flooded through the entire network resulting in a non-scalable number of messages exchanged for answering a query (Gnutella[9]). If the full state is kept locally, the data exchanged to maintain full state

and the memory needed for storing is non-scalable. We need to find a suitable routing schemes between these extremes.

In this thesis, two metrics are mostly used to evaluate the algorithm performance: cost of join/leave and lookup path length.

- **cost of join/leave**: The service should accommodate changes easily. In particular, when nodes join or leave, only a small number of nodes should change their state.

- **lookup path length**: The forwarding path of a lookup should involve as few nodes as possible. Most systems aim to minimize the maximum path length by applying overlay topologies with logarithmic diameters.

Earlier P2P systems employ a single index server or flooding-based mechanism to search desired data, which are not suitable for large systems. The latest P2P systems apply distributed hash table (DHT) to support fast data locating and system scalability. A very popular P2P system based on distributed hash tables is Chord[1]. Chord is a distributed lookup protocol designed to support fast data locating. It specifies how to find the location of keys, how new nodes join the system, and how to recover from failures or planned departures of existing nodes.

The objective of this thesis is to enhance the DHT based Chord system through some improvements. We will discuss the three aspects: routing, scalability and robustness of the enhanced system.

1.4 Thesis Contributions

A fundamental problem that confronts peer-to-peer applications is to efficiently locate the node that stores a particular data item. This thesis reviews the distributed hash table based P2P systems and presents three improvement systems based on Chord system to gain better routing and higher fault tolerance with very small extra cost of local storage and maintaining.

1.4.1 K-Chord Model

The k-Chord model applies k overlayed Chord rings to reduce the distance between the source node and the destination node sharply within the first few hops. The idea is 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.

In the system, each node has k identifiers and every identifier corresponds to a location in one Chord ring. Each data item has a unique key and is mapped into the same location in different Chord rings. Thus, there are k replicas of one data item distributed in k different nodes located on different Chord rings in the overlay network. The k distributed replicas of each data item are used to improve the routing performance and enhance the data availability.

1.4.2 Successor List Model

A successor list contributes a lot for efficient routing, failure recovery and data replication. In the successor list model, each node maintains a successor list of size d, containing the

node's first d successors.

Since Chord routing algorithm can halve the distance to the destination at each hop, the last several lookup hops may be within a very short distance to the destination. The successor list model uses the successor list of each node as part of the routing table to improve routing performance. It can merge the last several hops into one hop if the destination located within the successor list's distance. The successor list of each node is also used to store replicas of each data item. This placement of replicas means that, after a data item's successor fails, the data item is immediately available at the data item's new successor.

1.4.3 Hybrid Model

Since the k-Chord model contributes a lot to approach the destination rapidly within the first few hops, and the successor list model is helpful for getting quickly to the destination in the subsequent hops, the hybrid model merges these two models together to cumulate the performance improvement.

In this model, each node maintains a k-dimensional finger table and a successor list of size d for efficient routing. During a lookup, each intermediate node resolves the query and chooses the node that is numerically closest to the destination as the next hop node. The experiments show that the hybrid model cuts the routing path length of Chord system in half.

In the hybrid model, replicas of a data item are stored in the same location on k different Chords associate with its key and the d successors. The priority of lookup for

a data item is routed first to the numerically closest destination, and if the target node has failed, it re-routes the request to that node's successor directly. If all d successor nodes have failed, it abandons the current Chord ring, and re-routes the request to the numerical closest Chord among the remaining Chords. The process is continued until it finds the desired data or all the nodes storing the replicas of the data have failed. The search mechanism integrates the k-Chord system's fast approaching feature and the easy re-routing of the successor replication scheme when target failures occur.

1.5 Thesis Outline

This thesis presents three enhanced models for Chord to improve the routing performance and data availability. Chapter 1 gives the fundamental definitions and background of the P2P system, Chapter 2 reviews some typical P2P systems, Chapter 3 describes Chord system in detail, Chapter 4 presents the k-Chord system, Chapter 5 presents successor list model, Chapter 6 merges the former two models, and finally, Chapter 7 summarizes the thesis and proposes some future research work.

Chapter 2

Review of Literature

In this chapter, we review earlier data lookup approaches of P2P systems, and then discuss the latest distributed hash table based P2P systems with several sample systems.

2.1 Overview of First Generation P2P Systems

The first generation P2P systems employ a single index server or broadcast-based mechanism to search desired data, an approach not suitable for large systems. Some other decentralized P2P systems use hierarchical structures or an hybrid of centralized and decentralized structures.

2.1.1 Centralized Systems

Centralized systems apply a central server to store all the data address information. When a node makes a request for a data item, it searches at the server which returns the desired data address, then access the node that stores the data directly.

The typical system is Napster. It works by operating a centralized index server that directs traffic between individual registered users. Each time a user submits a request for a song, the central server creates a list of users who are currently connected to Napster

whose collections include the specified song.

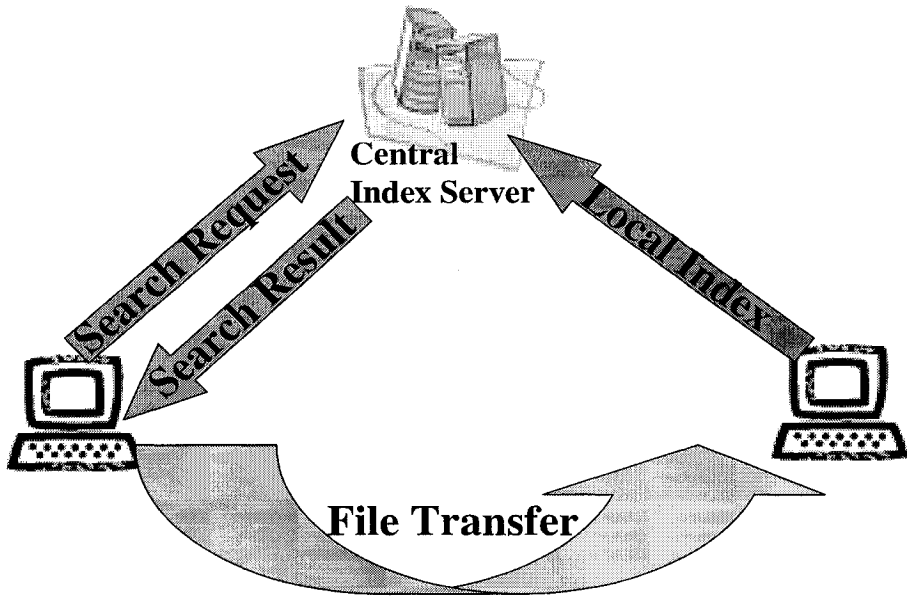

Figure 2.1: Napster

Although centralized P2P systems are simple and easy to manage all data information, they have some obvious problems that are not suitable for large scale systems.

- With a single failure of the central server the system becomes unusable
- The communication traffic and the storage on serverre very high

2.1.2 Decentralized Systems

Earlier decentralized P2P systems are not based on a central server. Most of them apply a broadcasting mechanism to search the desired data. When a node receive a request for a key that represents the data item, it attempts to retrieve the file locally if possible; otherwise, it forwards the request to another node. When a request is successful (or fails), the desired data item (or the failure report) is returned to the requester along the same

path of the incoming request. Some systems enhance the routing performance by caching or forwarding requests to neighbors that are more likely to store the desired data. The local forwarding decision may depend on former routing information or maintaining extra data index information.

Typical systems are, for example Gnutella[9] and Freenet[16]. Gnutella applies flooding-based broadcasting mechanisms, and Freenet uses backtracking approach to search for an object.

The disadvantages of these system are:

- No guaranteed reliable content locating information
- Expensive cost for routing(flooded mechanism) and bad scalability

2.2 Second Generation P2P Systems

A fundamental problem of practical distributed P2P systems is the efficient discovery and location of data and resources in a dynamic network. Most latest P2P systems, including Chord[1], Content-Addressable Networks (CAN)[2], D2B[5], Tapestry[6], Pastry[7] etc., use distributed hash tables(DHT) to support scalability, load balance and fault tolerance. At an abstract level, these systems all implement or 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 any stored resource by keys.

2.2.1 Distributed Hash Tables (DHTs)

A lookup service is a basic necessity in overlay networks. A good example is the domain name system (DNS) mapping host and domain names to IP addresses, which is static and suffers from congestion problems at the root of the DNS tree. In contrast, DHTs build a completely distributed and scalable lookup service suitable for deployment in peer-to-peer networks, where the participating nodes are particularly dynamic and no central control or information is easily maintained. The distributed hash tables implement storage and lookup of (key, value) pairs on a dynamic network.

There are two main challenges that large-scale DHTs must overcome: the first is distributing data in such a way that remains nearly balanced across the set of active nodes, and requires only small changes when nodes join and leave. The second is to maintain a network of connection information between nodes so that a lookup for data can be routed between nodes toward its intended target, and so that nodes may join and leave without requiring hash information to be propagated through the entire network.

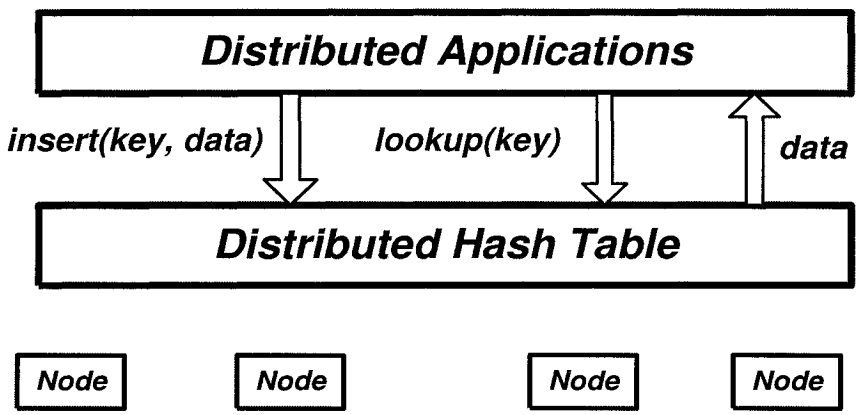

Figure 2.2: Distributed Hash Table

In DHT-based P2P systems, files are associated to keys(produced by hashing the file name or content), and identifiers of nodes are drawn from the same name space. Each node in the system handles a portion of the hash space and is responsible for storing a certain range of keys. After a lookup for a certain key, the system will return the identity (e.g., the IP address) of the node storing the object with that key.

DHT allows nodes to put and get files based on their keys, and has been proven to be a useful substrate for large distributed systems; a number of projects are proposing to build Internet-scale facilities layered above DHTs.

When a new node arrives, it joins the system through any node that is already in the system. The new joined node creates its routing table and is assigned the keys it is responsible for. When an old node leaves, it notifies other nodes of its departure, and transfers the keys it is in charge of.

In DHTs, each node handles a portion of the hash space and is responsible for a certain key range. Routing is a location-deterministic distributed lookup. The systems based on DHT have the following desirable characteristics:

- No global knowledge is needed
- A single fault does not disrupt the entire system
- Greater scalability
- Uniform distribution of resources

Thus, DHTs can use a fixed hash function to map both keys and nodes to the same

metric space; store key to object mappings at the node which is closest to the key (for a given metric); route the query message towards the node that is closest to the key.

2.2.2 Some DHT-based P2P Systems

Plaxton

Plaxton[14] (1997) presents a distributed data structure, called a Plaxton mesh, optimized to support locating and routing to named objects. It does not support dynamic node insertion or deletion and node failure handling. In Plaxton, objects and nodes have identifiers independent of their location and semantic properties, in the form of random fixed-length bit-sequences represented by a common base. The system assumes entries are roughly evenly distributed in both node and object name spaces, which can be achieved by using hashing algorithms.

Structuring: Each node X maintains a neighbor table and a pointer list of the node. The neighbor table consists of $(\log n)/b$ levels, and each level represents a matching suffix up to a digit position in the ID. A given level of the neighbor table contains a number of entries equal to b, the base of the ID. The pointer list $Ptr(X)$ with pointers to copies of some objects in the network in form of $< A, Y, k >$ which represents node Y has a copy of object A with cost function value of k. At most one record associated with any object is maintained in $Ptr(X)$. For each object, it is always guaranteed that there is a node that maintains the location mapping for it.

Locating & Routing: During a location query, the client sends message destined for object A is initially routed towards A's root. At each step, if the message encounters

a node that contains the location mapping for A, it is immediately redirected to the server containing the object. If the message reaches the root, it is guaranteed to find a mapping for the location of A. The local neighbor table, also called neighbor maps, is used to incrementally route messages to the destination ID digit by digit. Digits are resolved from the right to the left, but the decision is an arbitrary one. This routing method guarantees that any existing unique node in the system will be found within at most $\log_b N$ logical hops, in a system with an N size name space using IDs of base b. A way to visualize this routing mechanism is that every destination node is the root node of its own tree, which is a unique spanning tree across all nodes. Any leaf can traverse a number of intermediate nodes en route to the root node.

Tapestry

Tapestry[6] (2000) is an overlay infrastructure designed as a routing and location layer in OceanStore[8]. Tapestry has the same naming, structuring, and core locating & routing scheme as Plaxton, and supports scalability and handling failures.

Structuring: The same as Plaxton, each node has a neighbor map, which is organized into routing levels, and each level contains entries that point to a set of nodes closest in network distance that matches the suffix for that level. Each node also maintains a back pointer list that points to nodes where it is referred as a neighbor. Instead of assigning a single root for an object in Plaxton, Tapestry uses a distributed algorithm, called Surrogate Routing, to incrementally compute a unique root node for a object; and moreover each object gets multiple root nodes through concatenating a small globally

constant sequence of salt values to each object ID, then hashing the result to identify the appropriate roots.

Locating & Routing: When locating an object, Tapestry performs the same hashing process with the target object ID, generating a set of roots to search. In Plaxton, there exist multiple copies of data, and each node en route to the root node only stores the location of the closest replica to it. Tapestry, however, stores locations of all such replicas to increase semantic flexibility. There are only some small modifications of routing mechanism for improving fault tolerance, e.g. in case of bad links encountered, routing can be continued by jumping to a random neighbor node.

Scalability: Node insertion is easily implemented through populating neighbor maps and neighbor notification, and node deletion is more trivial. Tapestry provides two introspective mechanisms to adapt to environmental changes. First, in order to adapt to the changes of network distance and connectivity, Tapestry nodes tune their neighbor pointers by running a refresher thread which uses network Pings to update network latency to each neighbor. Second, Tapestry presents an algorithm that detects query hotspots and offers suggestions on locations where the additional copies can significantly improve query response time.

Pastry

Pastry[7] (2001) is a generic peer-to-peer content location and routing system based on a self-organizing overlay network via Internet. It is completely decentralized, scalable, fault-resilient, and reliably routes a message to the live node with a nodeId numerically

closest to a key with that message, and supports dynamic data object insertion/deletion, and dynamic node join/departure.

Naming: Each Pastry node has an unique 128-bit nodeId, this nodeId is assigned randomly when a node joins the system by computing a cryptographic hash of the node's public key or its IP address. With this naming mechanism, it ensures that nodeIds are generated such that the resulting set of nodeIds is uniformly distributed in the nodeId space. Each data also has a 128-bit key. This key can be the original key, or generated by a hash function. The data is stored in the node whose id is numerically closest to its key.

Structuring: Each node maintains a routing table, a neighborhood set and a leaf set. A node's *Routing table* is organized into $\log n$ rows with $2b - 1$ entries each row, where n is the number of nodes and b is a configuration parameter with typical value. The ith row of the routing table contains the nodeIds and IP addresses of those nodes, whose nodeId shares the present node's nodeId in the first i digits but different in the $i+1$ digit. If there are more than $2b - 1$ qualified nodes, the closest $2b - 1$ nodes will be selected, according to proximity metric. *Neighborhood set* contains the nodeIds and IP addresses of the nodes that are closest to the present node. *Leaf set* contains the nodeIds and IP addresses of the half nodes with numerically closest larger nodeIds, and half nodes with numerically closest smaller nodeIds, relative to the present node's nodeId.

Locating & Routing: Given a message, the node first checks if the key falls within the range of nodeIds covered by its leaf set. If so, the message is forwarded directly to the destination node, namely the node in the leaf set whose nodeId is closest to the key.

If the key is not covered by leaf set, then the routing table is used and the message is forwarded to a node that shares a common prefix with the key by at least one more digit. In certain cases, it is possible that the appropriate entry in the routing table is empty or the associated node is not reachable, in which case the message is forwarded to a node that shares a prefix with the key at least as long as the present node, and is numerically closer to the key than the present node's nodeId. Such a node must be in the leaf set unless the message has already arrived at the node with numerically closest nodeId.

Content-Addressable Network (CAN)

CAN[2] (2001) is a distributed hash-based infrastructure that provides fast lookup functionality on Internet-like scales. In CAN, nodes are addressed by their IP addresses. Each object has a unique key K. A hash function assigns a d-dimensional vector $P = hash(K)$ for each key, which corresponds a point in d-dimensional torus space and the point indicates the virtual position for the object.

Topology: CAN maintains a d-dimensional virtual space on a d-torus. The virtual space is partitioned into many small d-dimensional zones, which are chunks of the entire hash table. Each node corresponds to one zone and stores the data that are mapped to this zone by hashing its key. In the d-dimensional coordinate space, two nodes are neighbors if their coordinate spans overlap along $d - 1$ dimensions and abut along one dimension. Each node knows the zones and IP addresses of its neighbors.

To store a pair $(key, value)$, the key K is deterministically mapped onto a point P in the coordinate space by using the hash function. The corresponding $(key, value)$ pair is

then stored at the node that owns the zone within which the point lies. To retrieve an entry corresponding to key K, any node can apply the same deterministic hash function to map K onto point P and then retrieve the corresponding value from the point P. If the point is not in the zone owned by the requesting node or its immediate neighbors, the request must be routed through the CAN infrastructure until it reaches the node in whose zone P lies.

Routing: For a given key, the virtual position will be calculated; then starting from any existing node, the query is passed through the neighbors until it finds the IP address of the target node. In a d-dimensional space, each node maintains $2d$ neighbors (at most $4d$, in fact) and the average routing path length is $(d/4)(n^{1/d})$ hops. If $d = (\log n)/2$, it will achieve the same scaling properties as Chord[1].

Scalability: CAN supports data insertion and deletion in $(d/4)(n^{1/d})$ hops, and nodes can join and leave dynamically. When a new node joins the network through any existing node, it use the routing mechanisms to find the deterministic point and the node whose current assigned zone covers the point will split the zone in half and retain half and hand the other half to the new node. The split is done by assuming a certain ordering of the dimensions in deciding along which dimension a zone is to be split, so that zones can be re-merged when nodes leave. The $(key, value)$ pairs from the half zone to be handed over are also transferred to the new node. Naturally, the neighbors of the split zone must be notified so that routing can include the new node, and the new node learns the information of its coordinate neighbor set from the previous occupant. This set is a subset of the previous occupant's neighbors, plus that occupant itself.

When a node leaves CAN, it will hand over its zone and the associated $(key, value)$ information to one of its neighbors. If the zone of one of the neighbors can be merged with the departing node's zone to produce a valid single zone, then this is done. If not, then the zone is handed to the neighbor whose current zone is smallest, and that node will then temporarily handle both zones. The average cost for a node's joining is $(d/4)(n^{1/d})$ hops; for a node's leaving, it is constant time.

Fault Tolerance: In CAN, a node copies its data to one or more of its neighbors. This is very useful for load balance and fault tolerance. It can detect and recover node failure automatically through each node's neighbors. Each node will periodically send messages to all of its neighbors giving its zone coordinates and a list of its neighbors and their zone coordinates. The prolonged absence of a message from a neighbor indicates its failure. Once a node detects its neighbor's failure, it will initiate the $TAKEOVER$ mechanism to ensure the zones they occupied are taken over by the remaining nodes. The nodes which hold the data will periodically refresh stale entries to refresh lost data.

D2B

Topology: D2B[5] (2003) is a content-addressable network, and the underlying static topology of the d-dimensional D2B, $d \geq 2$, is the *De Bruijn* graph $B(d, k)$, for $k \geq 1$. It is the directed graph whose nodes are all strings of length k on the alphabet $\{0, \ldots, d-1\}$, and there is an edge form any node $x_1 x_2 \ldots x_k$ to the d nodes $x_2 \ldots x_k \alpha$, for $\alpha = 0, \ldots, d-1$. There are loops around all nodes $\alpha \ldots \alpha$, $\alpha \in \{0, \ldots, d-1\}$.

Routing: $B(d, k)$ has d^k nodes, in-degree and out-degree d, and diameter k. Routing

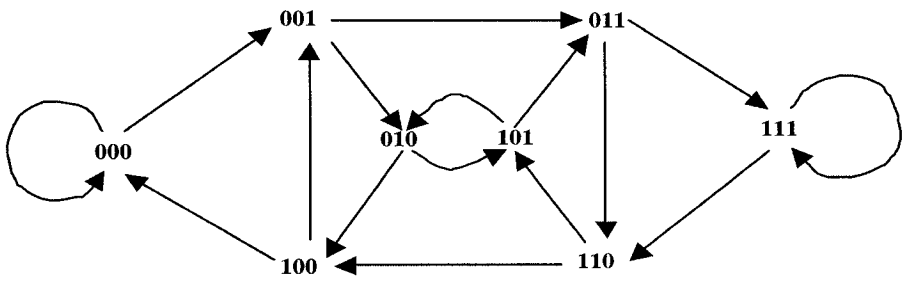

Figure 2.3: The de Bruijn graph $B(2,3)$

from $x_1 \dots x_k$ to $y_1 \dots y_k$ is achieved by following the route $x_1 \dots x_k \rightarrow x_2 \dots x_k y_1 \rightarrow x_3 \dots x_k y_1 y_2 \rightarrow \cdots \rightarrow x_k y_1 \dots y_{k-1} \rightarrow y_1 \dots y_k$. 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 $x_1 \dots x_k$, and prefix of $y_1 \dots y_k$. If there is such a sequence $x_i \dots x_k = y_1 \dots y_{k-i+1}$, then the shortest path from $x_1 \dots x_k$ to $y_1 \dots y_k$ is $x_1 \dots x_k \rightarrow x_2 \dots x_k y_{k-i+2} \rightarrow x_3 \dots x_k y_{k-i+2} y_{k-i+3} \rightarrow \cdots \rightarrow x_{i-1} \dots x_k y_{k-i+2} \dots y_{k-1} \rightarrow y_1 \dots y_k$. Here is an example, we want to route from 111000 to 000111. The path is $111000 \rightarrow 110001 \rightarrow 100011 \rightarrow 000111$.

Key Distribution: The 2-dimensional D2B uses the key name space $\mathbb{R} = \{0, \dots, 2^m - 1\}$ which is viewed as the set of binary strings of length m. The *value* of a node u labelled $x_1 \dots x_k, x_i \in \{0,1\}$, is $val(u) = 2^{m-k} \cdot \sum_{i=1}^{k} x_i 2^{k-i}$. Node u labelled $x_1 \dots x_k$ is responsible for all keys between $val(u)$ and $2^{m-k}(1 + 2^{m-k} \cdot \sum_{i=1}^{k} x_i 2^{k-i}) - 1$. In other words, the key whose binary representation is $\kappa_1 \dots \kappa_m$ is managed by node $x_1 \dots x_k$ currently in the system if and only if $x_1 \dots x_k$ is a prefix of $\kappa_1 \dots \kappa_m$. Hence, a node labelled $x_1 \dots x_k$ is responsible for 2^{m-k} keys. D2B maintains sibling connections for key redistribution.

Node Joining: When a node u joins the network through any existing node v, it is

assigned a temporary random label which is an m-bit string $s_1 \dots s_m$. The system uses longest prefix/suffix match method to route the joining message $< join, IP_u, s_1 \dots s_m >$ to the destination node ω with label $x_1 \dots x_k$ and responsible for the key $s_1 \dots s_m$. Therefore, $x_1 \dots x_k$ is a prefix of $s_1 \dots s_m$. If $k = m$, then the join fails and u must choose another temporary label. Node u gets the label $x_1 \dots x_k 1$, and ω extends its label to $x_1 \dots x_k 0$. The operation is called label extension. Node ω will transfer the keys with prefix $x_1 \dots x_k 1$ to the new joined node u.

The network also updates the connections of out-neighbors (children) and in-neighbors (parents). To update child-connections, for general case, if ω has a unique child labelled $x_2 \dots x_j, j \leq k$, then this child becomes the only child of both u and ω. If ω has several children, with labels of the form $x_2 \dots x_k y_1 \dots y_l, l \geq 1$, then those satisfying $y_1 = 1$ become the children of u, and ω will inform them that u will be their new parent; if children of ω with $y_1 = 0$, ω is still their parent. For specific case, if ω is labelled $\alpha \dots \alpha, \alpha \in \{0, 1\}$, then its children have the form $\alpha \dots \alpha y_1 \dots y_l, l \geq 1, y_1 = \overline{\alpha}$. By label extension, either ω or u takes label $\alpha \dots \alpha$, while the other takes label $\alpha \dots \alpha\overline{\alpha}$. Node labelled $\alpha \dots \alpha$ takes $\alpha \dots \alpha\overline{\alpha}$ as unique child, while node $\alpha \dots \alpha\overline{\alpha}$ takes all nodes $\alpha \dots \alpha y_1 \dots y_l = \alpha \dots \alpha\overline{\alpha} y_2 \dots y_l$ as children.

To update parent-connections, ω informs its every parent ω' the existence of the new joined node u. For general case, if ω' is labelled $\alpha x_1 \dots x_j, j \leq k$, then ω' takes u as one of its child, and modifies the label of ω in its routing table. Hence ω' has one more child, and its degree increases by one. If ω' is labelled $\beta x_1 \dots x_k y_1 \dots y_l$ with $l \geq 1$, then ω' keeps ω as child if $y_1 = 0$, or replace ω by u if $y_1 = 1$. For specific case, if ω is labelled

$\alpha \ldots \alpha, \alpha \in \{0,1\}$, by label extension, either ω or u takes label $\alpha \ldots \alpha\alpha$, while the other takes label $\alpha \ldots \alpha\overline{\alpha}$. If ω' has form $\overline{\alpha}\alpha \ldots \alpha$ with $(j \leq k)$ α's, then node labelled $\alpha \ldots \alpha$ takes ω' as its parents, while node labelled $\alpha \ldots \alpha\overline{\alpha}$ takes both $\alpha \ldots \alpha$ and ω' as parents. If ω' has form $\overline{\alpha}\alpha \ldots \alpha y_1 \ldots y_l$, with k α's and $l \geq 1$, the node labelled $\alpha \ldots \alpha$ takes those with $y_1 = \alpha$ as its parents, while $\alpha \ldots \alpha\overline{\alpha}$ takes those with $y_1 = \overline{\alpha}$, together with node $\alpha \ldots \alpha$ as parents.

Node Leaving: Considering node u labelled $x_1 \ldots x_k$ leaves the system, if a node v labelled $x_1 \ldots x_{k-1}\overline{x_k}$ exists, the lookup tables managed by u and v are merged and stored in v which is relabelled in $x_1 \ldots x_{k-1}$; if not, it uses label-extension to create a virtual binary tree rooted at $x_1 \ldots x_{k-1}\overline{x_k}$, whose leaves are nodes currently in the system. In this tree, the children of an internal vertex $x_1 \ldots x_{k-1}\overline{x_k}y_1 \ldots y_p$ are vertices $x_1 \ldots x_{k-1}\overline{x_k}y_1 \ldots y_p 0$ and $x_1 \ldots x_{k-1}\overline{x_k}y_1 \ldots y_p 1$. Since the depth of this virtual binary tree is finite, there is at least one pair of leaves whose labels differ only at the right most bit-position, called *critical pair*. One node of the critical pair will be the substitute for u, and the other will be the substitute for the two nodes of the critical pair.

To update the network, if the leaving node u belongs to the identified critical pair $\{v, v'\}$, then node u' labelled $x_1 \ldots x_{k-1}\overline{x_k}$ belongs to $\{v, v'\}$ as well, and becomes the substitute for u and u'. u' receives from u all information about the keys managed by u and the information about the sibling, parents, and children connections of u, and u' is relabelled in $x_1 \ldots x_{k-1}$. Node u' stores in its routing tables the IP-addresses and labels of the former children of u, which now become children of u'. Finally, u' informs the former parents of u that it is now their child, with label $x_1 \ldots x_{k-1}$.

If u does not belong to $\{v, v'\}$, assuming that v is labelled $x_1 \ldots x_{k-1}\overline{x_k}y_1 \ldots y_p 0$ and v' is labelled $x_1 \ldots x_{k-1}\overline{x_k}y_1 \ldots y_p 1$. Node v' is the substitute for v and v', while node v is the substitute for u. Hence v' perform the same procedure for v and v' as u' performed for u and u' in the previous case. In particular, the label of v' is contracted into $x_1 \ldots x_{k-1}\overline{x_k}y_1 \ldots y_p$. Node v takes the label of u, and retrieves from u its lookup and routing tables. As soon as v has retrieved all information from u, node u leaves the system.

2.2.3 Summary

Most DHT based P2P systems use different topologies, such as *mesh*, *torus*, *ring*, *de bruijn*, *butterfly* etc., to achieve better routing performance, better fault tolerance and less size of routing table. The most obvious measure of the efficiency of these routing algorithms is the routing path length, and most algorithms have path lengths of $O(\log n)$ hops, while CAN has longer paths of $O(dn^{\frac{1}{d}})$. The most obvious measure of the overhead associated with keeping routing tables is the number of neighbors. It is a measure of the state required to do routing and a measure of how much state needs to be adjusted when nodes join or leave. Most of the algorithms require $O(\log n)$ neighbors, while CAN requires only $O(d)$ neighbors. Most systems maintain a neighbor set for each node, and the neighbors communicate with each other periodically to detect and recover from failures.

We can summarize the features of DHT based systems in the Table 2.1 (the Join Cost including the cost for locating insertion position and the cost for creating routing table). Although most of these systems have the similar cost complexity, in contrast to them, our

hybrid system has some properties as simplicity, better performance (about $\frac{1}{4}\log n$ hops for lookup) and more robustness. Actually, the hybrid system inherits all the merits of Chord, and has better lookup performance and better fault tolerance.

	Topology	Routing	Local Storage	Join Cost	Leave Cost
Plaxton	Mesh	$O(\log n)$	$O(\log n)$	N/A	N/A
Tapestry	Mesh	$O(\log n)$	$O(\log n)$	$O(\log^2 n)$	$O(\log n)$
CAN	d-Torus	$O(dn^{\frac{1}{d}})$	$O(d)$	$O(dn^{\frac{1}{d}})$	$O(d)$
Chord	Ring	$O(\log n)$	$O(\log n)$	$O(\log^2 n)$	$O(1)$
CRN	Butterfly	$O(\log n)$	$O(\log n)$	$O(\log^2 n)$	$O(1)$
D2B	De Bruijn	$O(\log n)$	$O(1)$	$O(\log n)$	$O(1)$
Hybrid (this thesis)	K Rings	$O(\log n)$	$O(\log n)$	$O(\log^2 n)$	$O(1)$

Table 2.1: DHT-based P2P Systems

Chapter 3

Chord

Chord[1] is a distributed lookup protocol designed to support fast data locating and node joining/leaving. It specifies how to find the location of keys, how new nodes join the system, and how to recover from failures or planned departures of existing nodes. Chord is the basis of our system, therefore, we will describe it in detail.

3.1 Name Space and Topology

In Chord, both identifiers of nodes and data keys are drawn from the same m-bit name space. The space can be viewed as a circle, in which the highest identifier is followed by zero. Each node is assigned an m-bit node identifier, which is obtained by hashing its IP address. All possible $N = 2^m$ nodes are ordered in the one-dimensional circle; the nodes are mapped to this virtual circle according to their identifiers. For each identifier, the first node on its clockwise side is called its successor node.

Each data also has an unique key. Chord applies a consistent hashing function[12] to map each key onto the node whose identifier most closely follows the key on the identifier circle. Therefore, each node is responsible for the key range of (predecessor, its identifier],

which means each node in the system stores the data item whose key belong to the space range between the preceding existed node's identifier and its identifier.

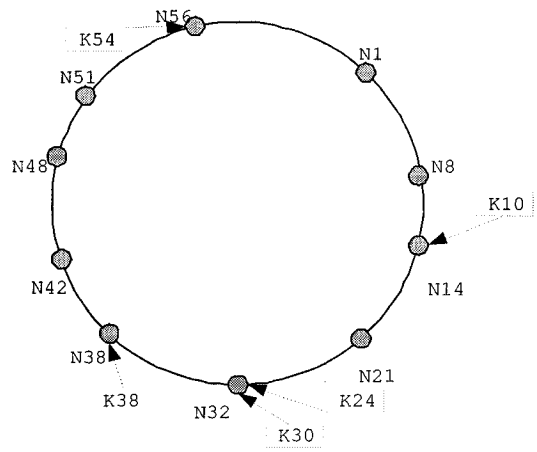

Figure 3.1: Key Distribution for Chord

Figure 3.1 shows a Chord ring with $m = 6$. The Chord ring has 10 nodes and stores five keys. The successor of identifier 10 is node 14, so key 10 would be located at node 14. Similarly, keys 24 and 30 would be located at node 32, key 38 at node 38, and key 54 at node 56.

Consistent hashing guarantees that nodes join and leave the network with minimal disruption. When a node u joins the network, certain keys previously assigned to u's successor now become assigned to u. When node u leaves the network, all of its assigned keys are reassigned to u's successor. No other changes in assignment of keys to nodes need occur. In the example above, if a node were to join with identifier 26, it would capture the key with identifier 24 from the node with identifier 32. Similarly, if node 56 leaves, the key with identifier 54 will be transferred from node 56 to its direct successor node 1.

Local State	Description
$finger[k]$	first node on circle that succeeds $(u + 2^{k-1}) \mod 2^m, 1 \leq k \leq m$
successor	the next node on the identifier circle, $finger[1]$.node
predecessor	the previous node on the identifier circle

Table 3.1: Local State for Each Chord Node

3.2 Finger Table and Routing

To do routing efficiently, each node contains the mapping information about other $\log N$ nodes. In the view of each node, the virtual circle is partitioned into $\log N$ segments, and the interval locations has the distances $2^0, 2^1, 2^2, \ldots, 2^{(\log N)-1}$ from it. The node maintains a finger table with $\log N$ entries; each entry contains the information for one segment: the boundaries and the successor of its first virtual node. In this way, each node only needs $O(\log N)$ memory to maintain the topology information, and the information is sufficient for fast locating and routing.

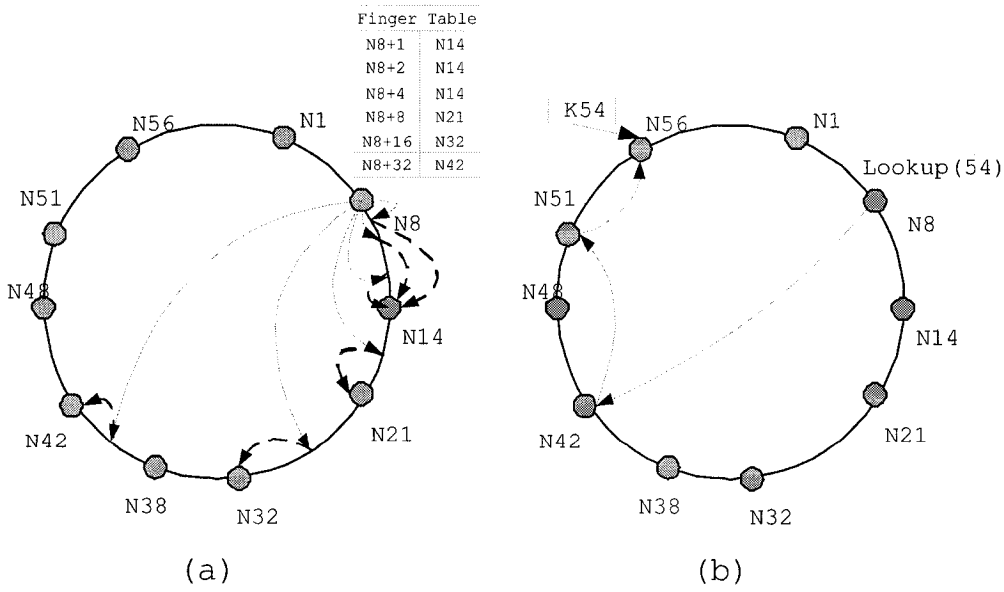

Figure 3.2: (a) Finger Table and (b) Routing for Chord

Table 3.1 indicates the local stored information for each node. Figure 3.2(a) shows the finger table of node 8. The first finger of node 8 points to node 14, as node 14 is the first node that succeeds $(8 + 2^0) \mod 2^6 = 9$. Similarly, the last finger of node 8 points to node 42, as node 42 is the first node that succeeds $(8 + 2^5) \mod 2^6 = 40$.

On query for a data item with key k, the virtual position is calculated firstly: $p = hash(k)$. The locating can start from any existing node. Using the finger mapping table, the successor of the segment that contains p is selected to be the next router until p lies between the start of the segment and the successor (this means the successor is also p's successor, i.e., the target). Since each node has finger entries at power of two intervals around the identifier circle, each node can forward a query at least halfway along the remaining distance between the node and the target identifier, which means the distance between the target and the current node will decrease by half after each hop. Thus, the routing time is $O(\log N)$.

```
//the pseudocode for finding the successor node of identifier ID
//ask node u to find the successor of ID
u.find_successor(ID)
    if (ID belong to (u, successor])
        return successor;
    else
        u' = closest_preceding_node(ID);
        return u'.find_successor(ID)

//search the local table for the closest predecessor of ID
u.closest_preceding_node(ID)
    for i = m downto 1
        if (finger[i] belong to (u, ID))
            return finger[i];
    return u;
```

The pseudocode of the *find_successor* operation implements the search process for a

desired ID through finger tables. If ID falls between node u and its successor, $find\_successor$ is finished and node u returns its successor. Otherwise, u searches its finger table for the node u' whose identifier most closely precedes ID, and then invokes $find\_successor$ at u'. The reason behind this choice of u' is that the closer u' is to ID, the more it will know about the identifier circle in the region of ID.

Figure 3.2(b) shows a example where node 8 wants to find the successor of key 54. Since the largest finger of node 8 that precedes 54 is node 42, node 8 will ask node 42 to resolve the query. In turn, node 42 will determine the largest finger in its finger table that precedes 54, i.e., node 51. Finally, node 51 will discover that its own successor, node 56, succeeds key 54, and thus will return node 56 to node 8.

3.3 Scalability

In Chord, nodes can join and leave at any time. When a node joins the system, it will start from any node already in the system. Through the start node, the joining node can recursively call $join()$ function to find its immediate successor. The node will ask its successor for the successor's predecessor p, and decides whether p should be its successor instead. The stabilization function notifies its successor of its existence, giving the successor the chance to change its predecessor to it. The successor does this only if it knows of no closer predecessor. To guarantee a correct lookup, each node's successor must be up to date. This is done at each node by periodically calling $stabilize()$ and $fix\_fingers()$ functions to learn about the newly joined nodes and refreshes its finger table.

```
//the pseudocode for joining and stabilization
//node u joins a Chord ring through the existing node u'
u.join(u')
    predecessor = nil;
    successor = u'.find_successor(n);

//called periodically and verifies u's immediate
//successor, and tells the successor about u.
u.stabilize()
    x = successor.predecessor;
    if (x belong to (u, successor))
        successor = x;
    successor.notify(u);

u.notify(u')
    if (predecessor is nil or n' belong to (predecessor, u))
        predecessor = u';

//called periodically to refresh finger table entries.
u.fix_fingers()
    next = next + 1; //next stores the index of the next finger to fix.
    if (next > m)
        next = index of first non-trivial finger entry;
    finger[next] = find_successor(u + 2^(next-1));
```

When a node leaves, it transfers its keys to its successor before departing, and the node also notifies its predecessor and successor before leaving. Thus, the predecessor and the successor can refresh their predecessor or successor respectively. For normal node arrival and departure, the cost is $O(\log^2 N)$ with high probability, but in the worst case, the cost is $O(N)$, where N is the size of the circle name space.

Chord uses a consistent hashing function to distribute nodes and data keys uniformly on the Chord circle; in this way the load is assumed to be balanced. The cost of a lookup is, in the worst case, $O(\log N)$; however, when N is big enough it does not depend on N and is $O(\log n)$, where n is the number of nodes (in Chord it is typically $N = 2^{24}$).

Moreover, each node does not need to maintain finger table entries between the node and its successor. The number of those entries is $\log(\frac{N}{n})$ under the assumption of nodes being uniformly distributed, and the number of finger table entries is $\log N - \log(\frac{N}{n}) = \log n$. Thus, also the joining/leaving cost does not depend on N with high probability and is $O(\log^2 n)$.

3.4 Fault Tolerance and Replication

To increase robustness, each Chord node maintains a successor list of size r, containing the node's first r successors. If a node's immediate successor does not respond, the node can replace it with the next entry in its successor list. Thus, the node failure can be detected and recovered automatically if each node maintains a r-successor list and calls periodically the refreshing procedure . Increasing r makes the system more robust. In Chord, it is proven that if $r = \Omega(\log n)$, every node fails with probability 1/2, then with high probability, *find_successor* can return the closest living successor to the query key and the lookup time is $O(\log n)$.

The Cooperative File System(CFS)[3], a peer-to-peer storage system based on Chord, replicates each data item on k nodes to increase data availability. It places the replicas at the k nodes immediately after the data item's successor on the Chord ring. The system can easily find these nodes from Chord's r-entry successor list, where $r \geq k$. This placement of replicas means that, after a data item's successor fails, the data item is immediately available at the data item's new successor. Nodes close to each other on the Chord ring are not likely to be geographically close to each other, since a node's identifier is based on

a hash of its IP address. This guarantees that there is little linear failures on the Chord and the replicas on the successors are always available.

Chapter 4

K-Chord Model

In this chapter, we present a new model based on Chord, called k-Chord model, to improve the routing performance and data availability of Chord. First, we describe the topology of the system and discuss why it will work. Then, four naming approaches are described to permutate the name space (each k-Chord node maintains a k-dimensional finger table to support efficient routing). Finally, simulations are provided to show the routing performance and fault tolerance of k-Chord compared with Chord system.

4.1 Topology

In the k-Chord model, we overlay k Chord rings one top of the other to generate a virtual network to speedup data lookup and make the system more robust. The idea is 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.

In the system, 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.

Figure 4.1 shows a k-Chord topology for $k = 2$. In the 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.

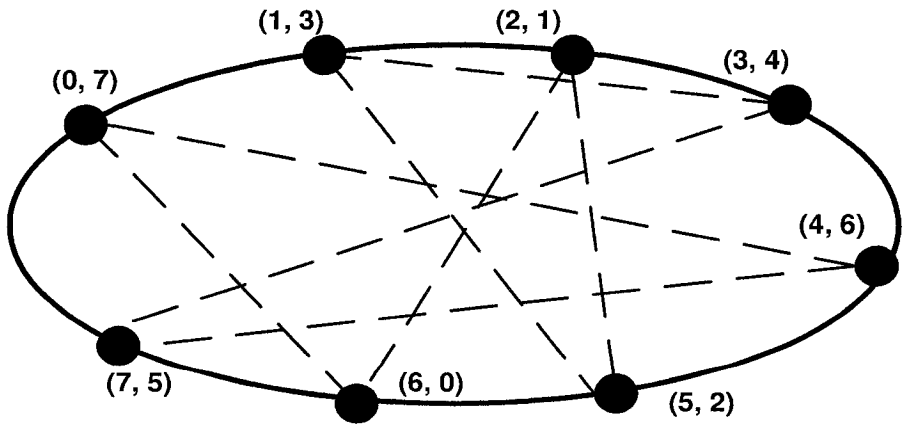

Figure 4.1: 2-Chord topology

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 one 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.

4.2 Permutation of k Name Spaces

In the k-Chord model, 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 $\mathbb{R}^0$. For example, we can see the two Chord rings in Figure 4.1 as the permutations:

0	1	2	3	4	5	6	7
7	3	1	4	6	2	0	5

A well-chosen permutation could make routing more efficient. In the following subsections, we offer four simple and practical permutations to accomplish the naming functionality.

4.2.1 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 $Id_2 = N - 1 - Id_1$.

4.2.2 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 view of the entire name space, 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 have constructed k-Chord system based 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.

4.2.3 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 record the 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 have obtained 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 maps different keys to distinct integers and has the same number of possible integers as keys, which means that n keys will map to $0..n-1$ with no collisions at all. Assuming 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 are different with high probability.

4.2.4 Modular Permutation

A modular approach can be used to obtain permutations that can cover the name space. Assuming that the size of N is a prime number and m is a 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 is co-prime with N, 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 $i = 0$.

For instance, with $k = 2, N = 11, m = 3$, and $i = 0$, we can get the following mapping table:

0	1	2	3	4	5	6	7	8	9	10
0	3	6	9	1	4	7	10	2	5	8

4.3 Routing Table and Locating

Locating and routing a desired data item efficiently is an extremely important issue in a P2P system. Chord can route to the destination node by decreasing the distance by half after each hop. In k-Chord system, we leverage k virtual Chords to speedup the routing

process. Each node will set up a finger table for each Chord ring respectively, which is exactly the same as Chord, and maintain the 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; if not, it will apply the greedy strategy to forward the query. The greedy algorithm 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.

```
Pseudo code for the greedy switch algorithm:
ClosestPredecessor()
{
    Distance of Current Predecessor = infinite;
    for (each finger table on k Chord ring) {
        if (target located within the successor position) {
            Jump to the destination node directly on this Chord ring;
            Lookup Done.
        }
        else    {
            Find closest predecessor on this Chord ring;
            Calculate the distance between it and the destination;
            if (the distance smaller then former distance)   {
                Distance of Current Predecessor = this distance;
                Record this predecessor and its Chord ring number;
            }
        }
    }
    Jump to final closest predecessor on its Chord ring;
}
```

We will see from the experiments that the approach of overlaying k topologies to improve performance is case sensitive. If the interval between each hop is 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 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 k Chords. It means that k-Chord model can contribute a lot in the first several hops by switching between different Chords, and does not help much for the subsequent hops.

4.4 Scalability

The k-Chord model supports the scalability well. For a new node to join or an existing node to leave, the cost is the same as Chord in each of the k Chord rings. Thus, the joining/leaving cost is k times the one of Chord, $O(k \log^2 n)$. Since k is a constant, the cost is $O(\log^2 n)$, the same as Chord. The experiment shows that applying a very small constant value of k, say $k = 4$, one can achieve a good routing performance.

4.4.1 Node Joining

When a new node joins the network, it can start from any node already in the system. Through the start node, the joining node can recursively call a $join()$ function (see Chapter 3) to find its immediate successor (same as Chord lookup procedure) on the k Chord rings or create a new network if it is the first node to join the system. The newly joined node will construct the connection to its predecessor and successor on each Chord ring,

and create the k-dimensional finger table. Its successors on each Chord ring will transfer back the data associated to the keys that precedes the new node.

To guarantee the correct lookup process, each node will refresh its k-dimensional finger table periodically to ensure that the successor pointers on each Chord ring are up to date with the evolution of the network. Each node runs *stabilize*() function (see Chapter 3) periodically to learn about the newly joined nodes and adjusts its successor and predecessor pointers on every Chord ring.

4.4.2 Node Leaving

When a node leaves, firstly, it will transfer its keys to its successor in every Chord ring. Secondly, the leaving node notifies its predecessor and successor to adjust their corresponding successor and predecessor pointers on the k Chord rings before leaving. Based on the above periodical running functions, the nodes that own the leaving node information in their k-dimensional finger tables will find the successor of the leaving node to substitute it.

4.5 Fault Tolerance

The failures fall into two categories from the view of a lookup: random failures and target failures. Random failures happen accidently and affect the routing procedure on the intermediate nodes during the lookup path; target failures only happen on the nodes that store the desired data, which will affect the routing at the end of the lookup path. We use different algorithms to process them.

4.5.1 Random Failures

When a massive random failure happens, a lookup may encounter intermediate failure nodes in the lookup path. The objective of the fault tolerance algorithm is to bypass the failed nodes and continue to approach the destination.

In Chord, during each lookup iteration every node selects from its finger table the largest *alive* node (alive nodes are those that do not fail) that precedes the target key to perform the next jump, and continues with further routing. In the k-Chord model, we adjust the greedy algorithm to return the numerically closest *alive* node to the destination during each hop. It 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.

4.5.2 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 should store the data item.

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

A valid but not efficient re-routing approach is that, when reaching a failed node that contains the desired data replica on one Chord, it will continue to look for the second closest location ignoring the Chord that contains the failure node, which is now considered an invalid Chord. The routing message will contain the previously found failure nodes information, and the lookup hops will only switch between the valid Chords. The obvious disadvantage of the scheme is that, as the number of target node failures increase, the number of available Chords decreases along with the lookup performance. When there are $k-1$ failure nodes that store the data item, it becomes identical to the Chord system.

4.6 Simulations

As described in Chapter 2, one of the purpose of DHT-based P2P systems is load balance. Thus, all our simulations are based on the uniform distribution of nodes in a Chord ring.

4.6.1 Lookup Performance

In this subsection, we show the lookup performance of the k-Chord model for all four permutation schemes considered in this chapter. In the experiments, the size of circle name space is $N = 10^6$, the number of nodes n is varied from 100 to 18000, and the number of k is varied from 1(Chord) to 6. The n nodes are hashed by their random generated IDs and distributed uniformly on the Chord ring. For each case, we choose 200 pairs of valid source nodes and desired keys randomly, and apply the average value of lookup path length(hops) to evaluate the lookup performance.

Reverse Permutation

Table 4.1 shows the average number of hops for reverse permutation. The improvement lookup performance for $k = 2$ is significant.

n	$k = 1$	$k = 2$
100	4.1	3.0
1000	5.8	4.6
5000	6.8	5.6
10000	7.4	6.0
12000	7.5	6.2
14000	7.5	6.4
18000	7.7	6.7

Table 4.1: Lookup Hops for Reverse Permutation

Shift Permutation

Table 4.2 and Figure 4.2 indicate the average number of hops for different choices of n and k. Increasing k, the lookup path lengths decrease. For $k = 2$, we get a significant performance improvement. Increasing k, the improvement slows down. When $k \geq 5$, the improvement is insignificant. Note that all the jump switches happened only on the first hop for 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 is better than that of the shift permutation. 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($k = 1$)

n	$k=1$	$k=2$	$k=3$	$k=4$
100	4.1	3.7	3.4	3.2
1000	5.8	5.4	4.8	4.5
5000	6.8	6.2	5.8	5.6
10000	7.4	6.7	6.4	6.1
12000	7.5	6.9	6.6	6.3
14000	7.5	7.0	6.7	6.5
18000	7.7	7.2	6.9	6.6

Table 4.2: Lookup Hops for Shift Permutation

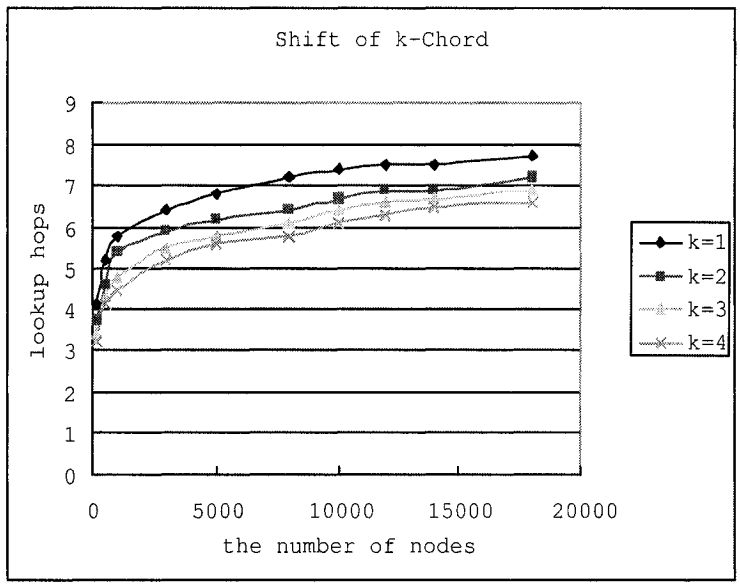

Figure 4.2: Lookup Hops for Shift Permutation

for each lookup. When $k > 2$, the average improvement by increasing k is only about 0.5 hop.

n	$k = 1$	$k = 2$	$k = 3$	$k = 4$
100	4.1	3.3	3.1	3.0
1000	5.8	5.0	4.8	4.3
5000	6.8	5.8	5.6	5.4
10000	7.4	6.1	5.9	5.7
12000	7.5	6.3	6.1	5.9
14000	7.5	6.5	6.2	6.0
18000	7.7	6.9	6.4	6.2

Table 4.3: Lookup Hops for Random Permutation

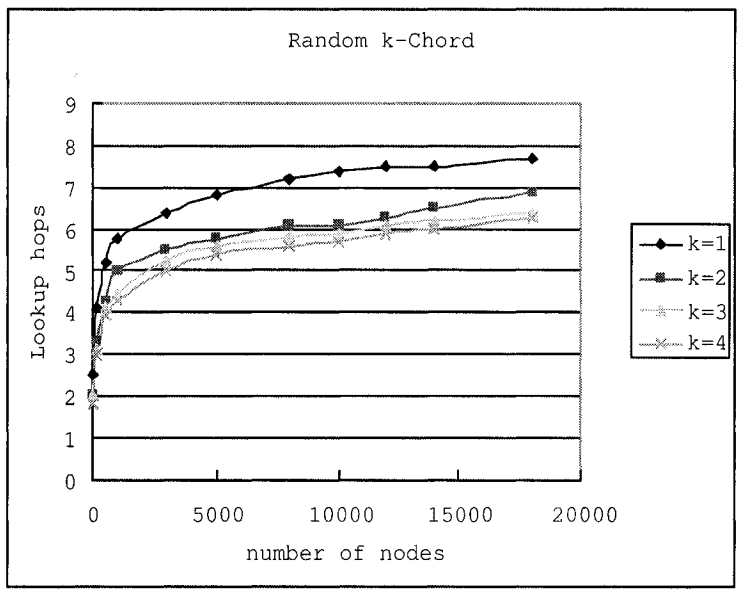

Figure 4.3: Lookup Hops for Random Permutation

Modular Permutation

In the test, 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 of m, we chose different m as: $m \approx (0, \log N, \sqrt{N}, \frac{N}{4}, \frac{N}{2}, \frac{3N}{4}, N)$ respectively and varied n varies from 1000 to 15000.

In the following experiment we have chosen to set $k = 2$ because when $k = 2$ there is a substantial improvement in comparison to Chord (i.e. $k = 1$). In fact, the average improvement is $0.8 \sim 1.3$ drop in hops for $k = 2$. However, increasing k, the drop becomes less and less significant. Up to $k = 4$, the improvement is still good; for $k > 4$, the improvement becomes irrelevant compared to the overhead. It is very interesting to notice that the choice of m does not have a significant impact on the performance of data lookup for modular permutation.

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

Table 4.4: Effect of Different m for Modular Permutation, $k = 2$

Since different m gives similar performance, we randomly chose m for the following experiments. Table 4.5 shows the simulation result. Again, as for the previous permutations we can observe that the performances increase significantly from $k = 1$ to $k = 2$ while the subsequent improvement is very low.

n	$k=1$	$k=2$	$k=3$	$k=4$
100	4.1	3.4	3.1	3.0
1000	5.8	5.0	4.6	4.3
5000	6.8	5.8	5.6	5.3
10000	7.4	6.2	5.9	5.7
12000	7.5	6.4	6.1	5.8
14000	7.5	6.5	6.2	6.0
18000	7.7	6.8	6.5	6.3

Table 4.5: Lookup Hops for Modular Permutation, m is random

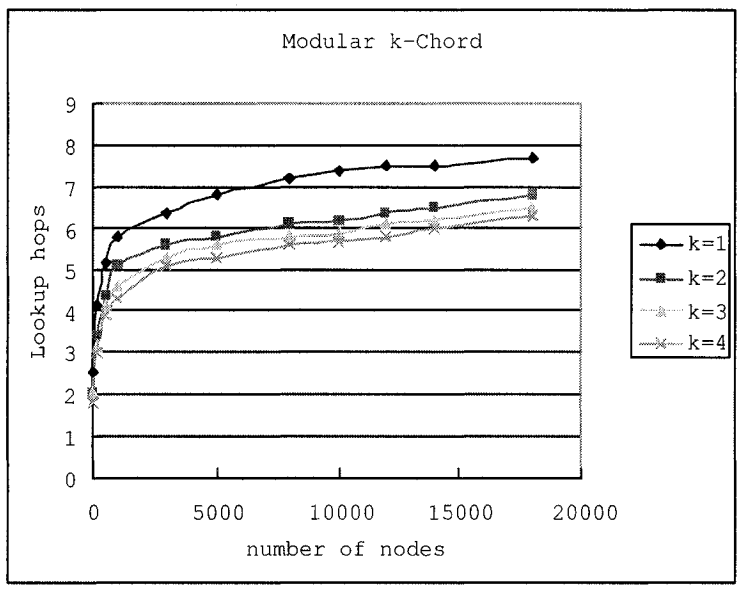

Figure 4.4: Lookup hops for Modular Permutation, m is random

Observations

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. Actually, it is not surprising that the modular scheme yields the same effect as the random permutation approach since many random number generators use modular scheme to generate random numbers.

The experiments of the four permutation schemes indicate that increasing the number of overlayed Chord rings the lookup path length decreases. However, increasing k also leads to the increase in overhead. Therefore, we need to find a trade-off between the performance improvement and the increasing overhead. The experiments show that when k varies from 1 to 2, the system achieves a significant improvement. Then, the improvement slows down. When $k \geq 5$, the improvement becomes insignificant. Thus, $k = 2, 3$ or 4 can be a good choice. If nodes join and leave the network frequently, $k = 2$ needs less extra overhead to improve the performance significantly; if the network does not change significantly, $k = 4$ achieves the best performance.

There is an open problem for the permutation: "Does there exist a better permutation for k-Chord model to achieve better routing performance?"

4.6.2 Simultaneous Node Failures

The Chord system applies a successor list for fault tolerance and failure recovery. The k-Chord model uses k replicas of a data item distributed on k Chord rings for fault tolerance. Since the k-Chord system does not maintain a successor list, it has little ability to recover

from failures. Here, we do not discuss the failure recovery and only see the lookup success rate and the lookup path length when failures happen.

To be comparable with Chord, we apply a similar setting in our experiments. There are 1000 nodes in the network, and 1000 data items with different keys are inserted into the network. Once the network becomes stable, each node is made to fail with independent probability p. After the failures occur, we do the test. In the simulation, we assume that the system is already in stable state to evaluate the performance. We randomly choose p fraction of all nodes failed and apply a status field in each node to show if it is alive. After that, we perform 1000 random lookups. For each lookup, we record the lookup path length if the lookup is successful. To see if the jumping node is alive, it simply checks the node's status information. If the lookup hops are more than a reasonable value, say 100, we consider that lookup to be unsuccessful. The experiment shows that Chord is better than k-Chord model for the fault tolerance and system recovery because of its successor list. In this case, the lookup success rate is a little bit low when massive failures occur(we will see an improved model in Chapter 6 and see that the improved model has better lookup performance when failure occurs).

4.7 Chapter Summary

In this chapter, we proposed a new model based on Chord called k-Chord model, which overlays several Chord rings to improve the routing performance and enhance the data availability. We described the organization of the k Chord rings and evaluated the routing performance through a simulation.

The following table shows the lookup performance of four permutations together with the lookup performance of Chord for comparison:

Permutation	k=1 (Chord)	k=2	k=3	k=4
Reverse	7.4	6.0	N/A	N/A
Shift	7.4	6.6	6.4	6.1
Random	7.4	6.1	5.9	5.7
Modular	7.4	6.2	5.9	5.7

Table 4.6: Lookup Performance of K-Chord, $n = 10^4$

Chapter 5

Successor List Model

In the successor list model, each node maintains a successor list of size d, where d is a constant, containing the node's first d successors. The successor list contributes a lot for efficient routing, failure recovery and data replication. Chord uses the successor list to increase system robustness, and CFS[3] uses it for data replication. Our enhancement is for efficient routing. In this chapter, we will demonstrate all these features in detail.

5.1 Successor List Routing

Since Chord routing algorithm can cut the distance to the destination by half at each hop, the last several lookup hops may be within a very short distance to the destination. Our idea is to use a 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 will check if the destination is located within the range of its position and the last node's position in its successor list instead of its direct successor. If the destination is located within that range, the node scans the successor

list to find the successor of 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.

```
Pseudo code for the routing algorithm:
ClosestPredecessor() {
    if (target located within the last successor position) {
        Find the successor of the looking key in the successor list;
        Jump to the destination node directly;
        Lookup done.
    }
    else    {
        Find closest predecessor in the finger table;
        Jump to that node to continue the lookup;
}
```

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. On the other hand, we choose a constant value for the length of the successor list. We will show that such a choice guarantees good routing performance.

5.2 Scalability

The successor routing model can support nodes joining and leaving exactly as in Chord. Compared to Chord, each node maintains the extra information of the successor list of size d as a part of its routing table. Since a node knows its successor and can access each successor in one hop, the cost for maintaining the list is $O(d)$. Since d is a constant, the extra cost is $O(1)$. Thus, the total joining/leaving cost is $O(\log^2 n)$, the same as Chord.

5.2.1 Node Joining

When a new node joins the network, it can start from any node already in the system. Through the start node, the joining node can recursively call the *join*() function (see Chapter 3) to find its immediate successor or create a new network if it is the first node to join the system. The new joined node will construct the connection to its predecessor and successor in the network to create the finger table and the successor list. Its successor will send back the data associated with the keys that belong to the new node.

To guarantee the correct lookup process, the join algorithm aggressively maintains the finger tables of all nodes as the network evolves and ensures the successor pointers are up to date. Every node periodically runs the *stabilize*() function (see Chapter 3) to learn about the newly joined nodes. The node asks its successor for the successor's predecessor p, and decides whether p should be its successor instead. This would be the case if node p recently joined the system. Also the *stabilize*() function notifies its successor of its existence, giving the successor the chance to change its predecessor to it. The successor does this only if it knows of no closer predecessor. Each node periodically calls the $fix\_finger()$ function (see Chapter 3) to make sure its finger table entries are correct; this is how new nodes initialize their finger tables, and how existing nodes incorporate new nodes into their finger tables. Each node periodically checks its predecessor to reset the node's predecessor pointer if its predecessor has failed; this allows the node to accept a new predecessor by notifying. A successor list refreshment is also done periodically.

5.2.2 Node Leaving

When a node leaves, it will do two things to ensure the correctness of the system. First, the leaving node u will transfer its keys to its successor before it departs. Second, it will notify its predecessor p and successor s before leaving. In turn, node p will remove the leaving node u from its successor list, and add the last node in the leaving node's successor list to its own list.

Similarly, the successor node s replaces its predecessor with the leaving node u's predecessor. Thus, the leaving node u sends its predecessor to its successor s, and the last node in its successor list to its predecessor p. Based on the above periodical running functions, the nodes that contain the leaving node information in their finger tables will find the successor of the leaving node to substitute it.

5.3 Fault Tolerance

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.

5.3.1 Failure Detection and Recovery

The successor list of a node can be used to detect and recover from failures automatically. The stabilization protocol is used to keep nodes' successor pointers up to date, which is sufficient to guarantee the correctness of lookups. 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 will periodically run the *stabilize*() function to do the recovery.

Successor lists are stabilized as follows: node u reconciles its list with its successor v by copying v's successor list, removing its last entry, and prepending v to it. If node u notices that its successor has failed, it replaces it with the first live entry in its successor list and reconciles its successor list with its new successor. At that point, node u can direct ordinary lookups for keys for which the failed node was the successor to the new successor.

The example in Figure 5.1 shows how the correcting scheme works. Assuming node 40 has successor list $(45, 52, 54, 61)$ and node 54 has successor list $(61, 67, 75, 85)$, when node 40 runs the *stabilize*() function and detects that its immediate successor and another node in its successor list fail, it will point to node 54 as its direct successor, which is the closest alive node in its successor list, and also notifies node 54 to change its predecessor pointer to node 40. Node 40 also reconciles node 54's successor list as its new successor list by dropping (82), which means node 40 adds two new closest nodes $(67, 75)$ into its successor list to substitute the two failed nodes and maintains the same size of the list.

It also transfers related information, such as the replicas of data if applicable, to the new nodes in successor list.

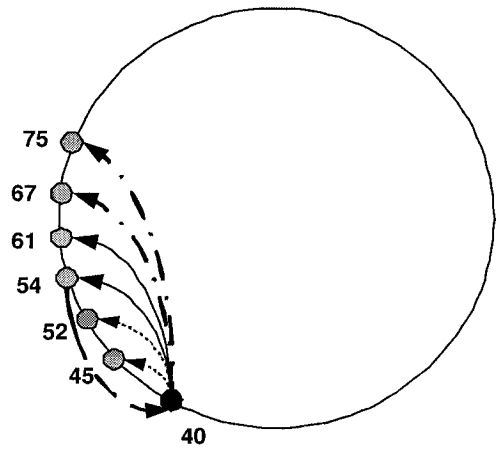

Figure 5.1: Recovery from the Failed Successor

We can simply add $fix\_fingers()$ into the periodically running scheme to refresh the finger table of each node. Each node checks every entry in its finger table; 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 finger table entries and successor list entries pointing to the failed node.

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 shows 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 can not 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 the data items are always available with high probability.

5.3.2 Random Failures

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.

5.3.3 Replication

The successor list mechanism also helps data replication and places replicas in a way that nodes can easily find them. We adopt the same replication scheme as CFS 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 keeps track of its r successors and propagates data to new replicas automatically when it detects that successors come and go.

Because 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. Thus, there is little linear failures and has high data availability with high probability. Actually, 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 have no control over the choice of the set on the logical Chord ring.

5.3.4 Target Failures

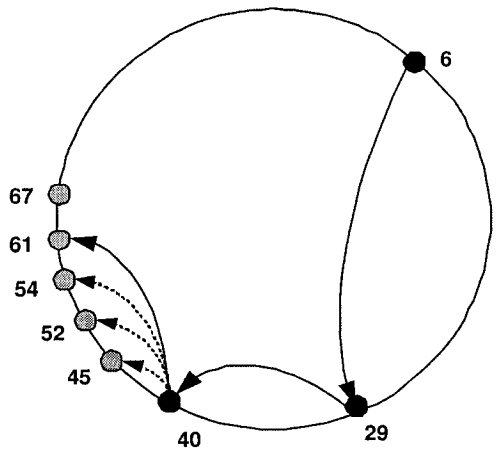

Figure 5.2: Replication of Successor List

Since replicas of a data item are stored at the r ($r \leq d$) nodes succeeding its key, when the node that stores this data fails, it can jump to its immediate successor to look for the desired data efficiently. In Figure 5.2, assuming node 40 has successor list $(45, 52, 54, 61, 67)$ and the desired data associating with key 51 which should be stored in node 52, so nodes 54, 61, 67 also own that data item's replicas. When a query reaches node 40, it checks the successor list of node 40 first and knows that node 52 should store the data item, so it forwards the query directly to node 52 without passing node 45, which

is the immediate successor of node 40. When node 52 fails, it re-directs to node 54 which also fails, then re-directs to node 61 which is available and stores the desired data item.

5.4 Simulations

5.4.1 Lookup Performance

In the experiment, the size of circle name space is $N = 10^6$, the number of nodes n is varied from 100 to 18000, and the successor list length d varies from 1(Chord) to 25. The n nodes are hashed by their random generated IDs and distributed uniformly on the Chord ring. For each case, we choose 200 pairs of valid source nodes and desired keys randomly, and apply the average value of lookup path length(hops) to evaluate the lookup performance.

We can see from Table 5.1 and Figure 5.3 that the system can achieve significant improvement for d up to 10. When $d > 20$, the performance improvement becomes insignificant.

n	$d = 1$	$d = 5$	$d = 10$	$d = 15$	$d = 20$	$d = 25$
100	4.1	3.2	2.7	2.3	2.5	2.4
1000	5.8	4.8	4.3	3.9	3.8	3.5
5000	6.8	5.8	5.3	5.2	5.0	4.8
10000	7.4	6.3	5.8	5.6	5.5	5.4
12000	7.5	6.4	5.9	5.7	5.6	5.4
14000	7.5	6.4	6.0	5.8	5.7	5.4
18000	7.6	6.7	6.1	6.0	5.8	5.6

Table 5.1: Lookup Hops of Successor List Model

The routing performance is better as we increase d. However, the increase of d will also increase the overhead. We need to find a trade-off value of d. The experiment shows

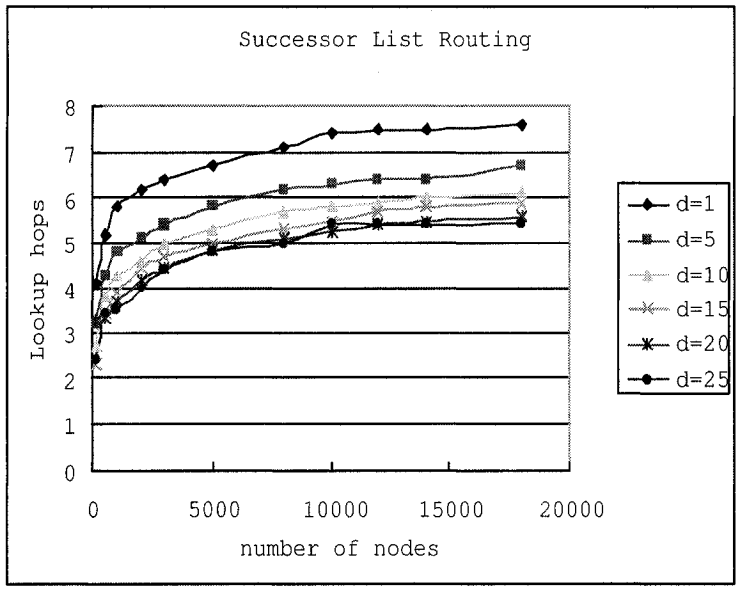

Figure 5.3: Lookup Hops of Successor List Model

that when $d > 20$, the improvement becomes insignificant compared to the overhead. From our experiments, we observe that $d = 20$ is sufficient for efficient routing. In the following we give an intuition of why $d = 20$ is a good choice in our setting. Assuming 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 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 about 128 long (when $i = 8$, the distance of $2^8 = 256$ will request at least 26 successors under 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 half of the distance to the destination, 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 routing list. Under this model, assuming the length of the successor list is $d = 20$, the experiment shows that the total decreased hops are about $1 \sim 2$.

5.4.2 Simultaneous Node Failures

Since the successor list system is used for fault recovery also in the Chord system, our system should have the same ability of the fault tolerance and system recovery. We do the test for 1000 nodes and 1000 data items with different keys in the network. The test shows that the successor list system has the same lookup success rate and a little bit shorter lookup path length than Chord, since the successor list system applies the successor list as part of the routing table. Table 5.2 shows the average lookup length when failures occur.

p(failure fraction)	0	10%	20%	30%	40%	50%
Chord $(r = 6, d = 20)$	5.8	6.0	6.2	6.5	6.8	7.2
Succ $(r = 6, d = 20)$	3.8	4.1	4.5	4.9	5.5	6.1

Table 5.2: The Successor List Model Lookup Path Length for Failures

5.5 Chapter Summary

In this chapter, we have extended the functionality of the Chord's successor list for efficient routing. The successor list of each node can be used for data replication, failure recovery, and fault tolerance. The experiments show that the successor routing approach helps significantly in data lookup.

The lookup performance of the successor list model can achieve notable routing im-

provement when the length d varies up to 10; there is nearly no improvement when $d > 20$. We consider that $d = 20$ is a good choice for the length of successor list. Table 5.3 reviews the routing performance of the successor list system with $d = 20$ and Chord ($d = 1$).

	$n = 1000$	$n = 5000$	$n = 10000$	$n = 15000$	$n = 18000$
Chord System	5.8	6.8	7.4	7.5	7.6
Successor List System	3.8	5.0	5.5	5.7	5.8

Table 5.3: Routing Performance compared with Chord

Chapter 6

Hybrid Model

Since the k-Chord model contributes a lot to approach the destination rapidly within the first few hops, and the successor routing model is helpful for getting quickly to the destination in the subsequent hops, we can merge these two models together to cumulate the performance improvement. The hybrid model combines the successor nodes and part of finger nodes to generate the routing table on each Chord ring. It helps speeding up the routing performance and fault tolerance.

6.1 Hybrid Routing

In the hybrid system, each node maintains a k dimensional finger table and a successor list of size d 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 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.

```
Pseudo code for routing:
ClosestPredecessor() {
    Distance of Current Predecessor = infinite;
    for (each finger table on k Chord ring) {
        if (target located within the last successor position) {
            Jump to the destination node directly on this Chord ring;
            Lookup Done.
        }
        else    {
            Find closest predecessor on this Chord ring;
            Calculate the distance between it and the destination;
            if (the distance smaller then former distance)   {
                Distance of Current Predecessor = this distance;
                Record this predecessor and its Chord ring number;
            }
        }
    }
    Jump to final closest predecessor on its Chord ring;
}
```

By overlapping the former two systems a potential problem could be that the hybrid system decreases the total performance improvement. Actually, the experimental results are better than our expectation. The k-Chord system reduces the distance between the source and the destination node sharply to 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 system 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 just overlapping the improvements of the former two systems. It is also helpful for fault tolerance and re-routing in the event of failures.

6.2 Scalability

Similarly, 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. Thus, the joining/leaving cost is $O(k(log^2 n + d))$. If we choose constants for k and d (e.g., $k = 4$ and $d = 20$), the total joining/leaving cost is $O(log^2 n)$, the same as Chord.

6.2.1 Node Joining

When a new node joins the network, it can start from any node already in the system. Through the start node, the joining node can recursively call the $join()$ function (described in Chapter 3) to find its immediate successor or to create a new network if it is the first node to join the system. The newly joined node will construct the connection to its predecessor and successor on each Chord ring, and create 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.

To guarantee the correct lookup process, every node runs the $stabilize()$ function (described in Chapter 3) periodically to refresh its k-dimensional finger table, and ensures that the successor list and successor pointers on each Chord ring are up to date with the

evolution of the network.

```
Pseudo code for stabilization:
//the function is run periodically by each node
Stabilize() {
    fix_fingers();  //refresh the finger table
    .......
}
```

6.2.2 Node Leaving

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 will be refreshed in the same way as former two systems.

6.3 Fault Tolerance

The replication schemes of both k-Chord and d-successor list can increase the data availability of the hybrid system. The successor list of each node helps the system to detect and recover from failures automatically, in the same way as in the successor list system.

6.3.1 Random failures

Before the system recovers from random failures, every node routes the request to the numerically closest *alive* node to the destination at each lookup hop. This node is chosen from both of the k-dimensional finger table and the successor list.

6.3.2 Target Failures

In the hybrid model, replicas of a data item are stored in the same location on k different Chords associate 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 has failed, it re-routes the request to that node's successor directly. If all d successor nodes fail, it abandons this Chord ring, and re-routes to the numerical closest Chord ring within the remaining valid Chord rings and does the same lookup mechanism until it finds the desired data or all the nodes storing the replicas of the data have failed. The search mechanism integrates the k-Chord model's fast approaching feature and the easy re-routing of the successor replication scheme when target failures occur.

We can compare the lookup performance of the hybrid model to that of the Chord when failures happen. Since all the replicas of a data item are distributed randomly, Chord and the hybrid system should have the same data availability. However, because of the distinct data replication and routing algorithm in the overlay network, there is a little bit difference in the performance.

Assuming there are m replicas of a data in both systems, Chord stores one replica in the node that is in charge of its key and $m-1$ replicas in its successors, whereas the hybrid system stores k replicas in k nodes that are in charge of its key and $\frac{(m-k)}{k}$ replicas in the succeeding nodes on k Chord rings. Obviously, if the linear failures in one Chord ring is less than $\frac{(m-k)}{k}$, the hybrid system has better re-routing performance for its more efficient lookup performance. If the first linear failures in one Chord ring is greater than

$\frac{(m-k)}{k}$, Chord is better. However, 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 the Chord system. We will see that in the simulation result in the following section.

6.4 Simulations

6.4.1 Lookup Performance

In the experiment, the size of circle name space is $N = 10^6$, the number of nodes n is varied from 100 to 18000: (i)to compare the effect of the successor list length for the hybrid system, the number of Chord rings is fixed $k = 2$, and the successor list length d varied from 1 to 25; (ii)to compare the effect of the number of Chord rings for the hybrid system, the successor list length is fixed $d = 20$, and the number of k varies from 2 to 7 where random permutation is applied for them. The n nodes are hashed by their random generated IDs and distributed uniformly on the Chord ring. For each result, we choose 200 pairs of valid source nodes and desired keys randomly, and apply the average value of lookup path length(hops) to evaluate the lookup performance.

n	$d = 1$	$d = 5$	$d = 10$	$d = 15$	$d = 20$	$d = 25$
100	3.4	2.5	2.2	1.9	1.8	1.7
1000	5.0	4.0	3.5	3.3	3.1	2.8
5000	6.0	5.2	4.5	4.2	3.9	3.7
10000	6.3	5.5	5.2	4.8	4.7	4.4
12000	6.5	5.6	5.4	5.0	4.8	4.6
14000	6.6	5.7	5.4	5.1	4.9	4.7
18000	6.9	6.1	5.5	5.2	5.0	4.8

Table 6.1: Lookup Hops for the Hybrid System, $k = 2$

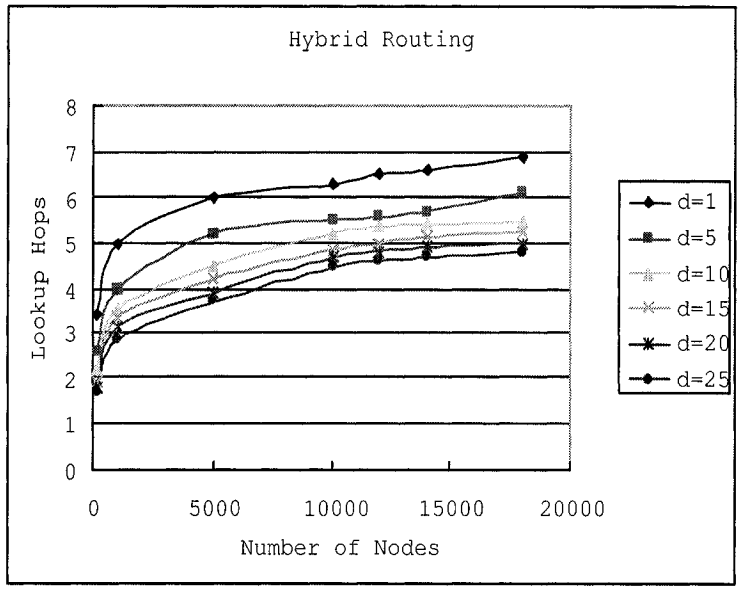

Figure 6.1: Lookup Hops for the Hybrid System, $k = 2$

n	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$	$k = 6$	$k = 7$
100	2.6	1.7	1.5	1.4	1.3	1.2	1.2
1000	3.8	3.0	2.7	2.5	2.5	2.3	2.2
5000	5.0	4.1	3.6	3.4	3.3	3.1	3.0
10000	5.5	4.4	4.1	3.9	3.7	3.5	3.4
12000	5.6	4.6	4.2	4.0	3.8	3.6	3.5
14000	5.7	4.7	4.3	4.1	3.9	3.7	3.6
18000	5.8	4.9	4.5	4.2	4.0	3.8	3.6

Table 6.2: Lookup Hops for Hybrid System, $d = 20$

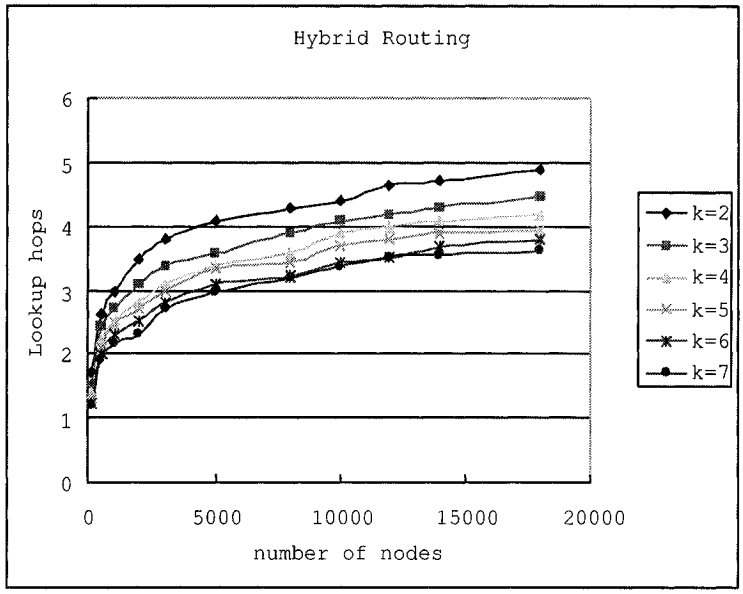

Figure 6.2: Lookup Hops of the Hybrid System, $d = 20$

Figure 6.2 shows that k varying from 1 to 4 achieves the best performance improvement. The experiment also indicates that when $k = 2$ and $d = 20$ it can gain very good routing performance and $k = 4$ halves the routing hops of Chord. All the results are based on uniform distribution.

6.4.2 Effect of Distribution Density

In Chord, the nodes are distributed uniformly on the Chord rings, hence the lookup performance depends only on the number of nodes in the network, which means 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 the simulations, the number of nodes $n = 10^4$ is fixed, the size of the name space N is varied from 20000 to 10^7, the number of Chord rings is varied from 1 to 4 and the successor list length $d = 20$ as shown in Table 6.3. The experiment indicates that increasing N, the

lookup path length increases very little. We conclude that under uniform distribution, the routing cost of the hybrid system depends mostly on the number of nodes in the system and not on the size of the name space.

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

Table 6.3: Effect of Distribution Density on the Hybrid System, $d = 20$

6.4.3 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 it, 1000 data items are inserted into a 1000-node system, and each data item has 6 replicas. For the value $k = 2$, $d = 20$, p fraction (varied from 10% to 50%) of all nodes are randomly chosen as failure nodes. After that, we perform 10000 random lookups. For each lookup, we record if the lookup has success and if it does, we calculate the lookup path length. Finally, we derive statistics of the lookup success rate and the average lookup path length(only for the success lookups). The experiments show that the hybrid system is better than the Chord system for the fault tolerance performance.

Table 6.4 shows the lookup failed rate when the failure fraction p varies from 10% to 50%, and r is the number of the successors that stores the data replicas and both systems have the successor list of size $d = 20$. 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 6.5 shows the average lookup path length when failures occur. The result indicates that the our hybrid system has better lookup performance when failures occur.

p(failure fraction)	10%	20%	30%	40%	50%
Chord $(k = 1, r = 6, d = 20)$	0	0	0.002	0.010	0.016
Hybrid $(k = 2, r = 3, d = 20)$	0	0	0	0.009	0.014

Table 6.4: Hybrid and Chord Lookup Failure Rate

p(failure fraction)	0	10%	20%	30%	40%	50%
Chord $(k = 1, r = 6, d = 20)$	5.8	6.0	6.2	6.5	6.8	7.2
Hybrid $(k = 2, r = 3, d = 20)$	3.1	3.5	4.1	4.8	5.6	6.6

Table 6.5: Hybrid and Chord Lookup Path Length for Failures

6.5 Chapter Summary

In this chapter, we have merged the k-Chord model and the successor list model together to generate a hybrid model. The hybrid model combines the improvements of both models to speedup routing and enhance fault tolerance.

The simulation (see Table 6.6) shows that the hybrid system can reduce the lookup hops to $\frac{1}{4}\log n$, which is half of the Chord system.

	$n = 1000$	$n = 5000$	$n = 10000$	$n = 15000$	$n = 18000$
Chord $(k = 1, d = 1)$	5.8	6.8	7.4	7.5	7.7
Hybrid $(k = 4, d = 20)$	2.5	3.4	3.9	4.1	4.2

Table 6.6: Hybrid Model and Chord Comparing

Chapter 7

Conclusion & Future Work

7.1 Summary

In this thesis, we have proposed three new overlay networks based on Chord with the objective of speeding up data lookup and enhancing fault tolerance. This goal has been achieved by overlaying k Chord rings and maintaining a successor list for every node on each Chord ring. The following sections summarize all the models and the results.

7.2 Models for the Overlay Networks

7.2.1 K-Chord Model

In the k-Chord model, the system overlays k Chord rings to speedup data lookup and make the system more robust. Among these Chord rings, one could choose, at each step of the lookup, the best Chord system to achieve better routing performance. When a query arrives at a node, the node scans the routing table which includes all finger table information for all k Chord rings, then routes to the numerically closest node which may store the data replicas. Our experiments show that the k-Chord system can reduce the distance between the source and the destination node sharply to within the first few hops.

Since the k-Chord system distributes k replicas of each data item on k different Chord rings, the robustness of the system is enhanced.

7.2.2 Successor List Model

In the successor list model, the system maintains a successor list of size d for each node as part of the routing table to improve routing performance. It merges the last several hops into one hop if the destination is located within the successor list's distance. This can be easily implemented by substituting the direct successor in Chord algorithm by the last successor in the successor list to check if the destination is located within the range and then routing the request to corresponding node directly.

The successor list model also applies the successor list for data replication and increasing system robustness. It stores the data replicas in the successors of the target node. When the node that has stored the desired data fails, the query can be forwarded directly to the immediate successor to look for the desired data efficiently.

7.2.3 Hybrid Model

The hybrid model combines the k-Chord model and the successor list model to cumulate the performance improvement. It applies both replication mechanisms of Chord and k-Chord to strengthen the robustness of the system. First, k-Chord system distributed the replicas on k different Chord rings to speedup searching the desired data. Second, it also stores the replicas in the successors of the target node. This approach is the same as Chord and can help it route to the immediate successor to look for the desired data efficiently if target failures happen.

If the total replica number of one data item is the same for both Chord and the hybrid system, which means the later system has less replicas on each Chord ring, the hybrid system may need more hops for full linear failures on each Chord ring. However, the probability of full linear failure is very low, and the total performance of the hybrid system is better than that of Chord.

The following Table 7.1 shows the three enhanced models and Chord together to compare the performance:

	$n = 1000$	$n = 5000$	$n = 10000$	$n = 15000$	$n = 18000$
Chord $k = 1, d = 1$	5.8	6.8	7.4	7.5	7.7
k-Chord $k = 4, d = 1$	4.3	5.4	5.7	6.0	6.2
Succ $k = 1, d = 20$	3.8	5.0	5.5	5.7	5.8
Hybrid $k = 4, d = 20$	2.5	3.4	3.9	4.1	4.2

Table 7.1: Global Summary of Experiment Results

The hybrid system has all the merits of Chord, such as simplicity, provable correctness and provable performance. Moreover, it has better lookup performance (halving the routing path length of Chord) and k Chord ring replication scheme enhances the data availability and fault tolerance. It maps its nodes to a logical one-dimensional space and replicates them on k Chord rings within which routing is carried out by an logarithmic complexity algorithm. It also has the advantage that its correctness is robust in the face of partially incorrect routing information.

7.3 Future Works

P2P systems based on distributed hashing able are the latest P2P research trends, which can support load balance, efficient locating and scalability well. The current research is focusing on using different topologies, such as *torus*, *ring*, *de bruijn*, *butterfly* and so on, to achieve better routing performance, better fault tolerance and smaller routing table.

Our future work will focus on overlaying some other homogeneous or heterogeneous topologies together to achieve better performance.

- *De Bruijn + Chord ring*

  This overlayed network applies *De Bruijn* topology for efficient routing, the same routing algorithm as D2B[5], and Chord ring topology for data replication, the same as Chord. A recently published paper[10] uses this overlay topology for optimal degree; in this system (called Koorde) each node maintains only two neighbors' information for routing.

- *k-De Bruijn*

  It is similar as our k-Chord system. Each node belongs to k *De Bruijn*s and has k names on each *De Bruijn*; each data item has one unique key. During a lookup, the best *De Bruijn* is chosen to route.The switch metric may be different from k-Chord system which applies numerically closest metric. Naturally, the update is equivalent to k *De Bruijn*'s updates. This solution would be good for efficiency and redundancy.

- *De Bruijn, Chord ring, Hypercube*, etc.

This overlay topology may combine all potential topologies together, and apply them for routing, replication, scalability and local storage.

- Mobile peer-to-peer system

  Some mobile networks are P2P systems, such as ad hoc network and sensor network. We can apply DHT mechanism to these system for efficient location.

All these possible systems need to solve following problems:

- What are all the information to be stored at each node.
- How to do the lookup, including the schemes for routing table and routing algorithm.
- How to maintain the topology, including the schemes for insertions and deletions.
- What is the performance.

Bibliography

[1] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications," *ACM SIGCOMM*, pp.149-160, 2001.

[2] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker, "A Scalable Content-Addressable Network," *Proceedings of ACM SIGCOMM*, pp.161-172, 2001.

[3] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica, "Wide-Area Cooperative Storage with CFS," *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[4] Jerry Banks, John S. Carson, B. L. Nelson, and D. M. Nicol, Discret-Event System Simulation, 3rd Edition, Prentice-Hall, 2001.

[5] P. Fraigniaud, P. Gauron, "The Content Addressable Network D2B", Technical Report LRI1349, University Paris-Sud, 2003.

[6] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph, "Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing", U. C. Berkeley Technical Report UCB//CSD-01-1141, 2000.

[7] A. Rowstron and P. Druschel, "Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems", Lecture Notes in Computer Science Vol.2218, pp.329-350, 2001.

[8] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, B. Zhao, "OceanStore: An Architecture for Global-Scale Persistent Storage," *In Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[9] Matei Ripeanu, Ian Foster, and Adriana Iamnitchi. "Mapping the Gnutella Network: Properties of Large-Scale Peer-to-Peer Systems and Implications for System Design," *IEEE Internet Computing Journal*, Vol.6, 2002.

[10] Frans Kaashoek, David R. Karger, "Koorde: A Simple Degree-optimal Hash Table," *In 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, pp.98-107, 2003.

[11] Sylvia Ratnasamy, Scott Shenker, Ion Stoica, "Routing Algorithms for DHTs: Some Open Questions," *In the proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS '02)*, pp.45-52, 2002.

[12] D. Karger, E. Lehman,F. Leighton, M. Levine, D. Lewin, AND R. Panigrahy,"Consistent hashing and random trees: Distributed caching protocols for re-

lieving hot spots on the World Wide Web" *In Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pp.654-663, 1997.

[13] Napster, http://www.napster.com/.

[14] C. Greg Plaxton, Rajmohan Rajaraman, Andra W. Richa. "Accessing Nearby Copies of Replicated Objects in a Distributed Environment," *Series-Proceeding-Article, ACM Press New York, NY, US*, pp.311- 320, 1997.

[15] TRIAD project, http://www.dsg.stanford.edu/triad.

[16] Ian Clarke et al, "Freenet: A Distributed Anonymous Information Storage and Retrieval System," Lecture Notes in Computer Science, 2000.

[17] D. Malkhi, M. Naor, D.Ratajckzac, "Viceroy: A Scalable and Dynamic Emulation of the Butterfly," *21st ACM Symposium on Principles of Distributed Computing (PODC)*, 2002.

[18] M. Datar, "Butterflies and Peer-to-Peer networks," *In European Symposium on Algorithms (ESA)*, 2461, pp.310, Springer, 2002.

[19] David Liben-Nowell, Hari Balakrishnan and David Karger, "Observations on the Dynamic Evolution of Peer-to-peer Networks," *In the proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS '02)*, pp.22-33, 2002

[20] Fiat et al, "Censorship resistant peer-to-peer content addressable networks," *ACM/SIAM Symposium on Discrete Algorithms (SODA)*, pp.94-103, 2002.

[21] Nancy Lynch, Dahlia Malkhi and David Ratajczak, "Atomic Data Access in Content Addressable Networks," *In the proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS '02)*, pp.295-305, 2002

[22] Petar Maymounkov and David Mazieres, "Kademlia: A Peer-to-peer Information System Based on the XOR Metric," *In the proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS '02)*, pp.53-65, 2002.

[23] Emil Sit and Robert T. Morris, "Security Considerations for Peer-to-Peer Distributed Hash Tables, *In the proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS '02)*, pp.261-269, 2002.

[24] David R. Karger and Matthias Ruhl, "Finding Nearest Neighbors in Growth-restricted Metrics," *ACM Symposium on Theory of Computing (STOC)*, 2002.

[25] Udi Wieder, Moni Naor, "A Simple Fault Tolerant Distributed Hash Table," *In 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, pp.88-97, 2003.

[26] Ranjita Bhagwan, Stefan Savage, Geoffrey Voelker. "Understanding Availability," *In 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, pp.256-267, 2003

[27] Indranil Gupta, Kenneth Birman, Prakash Linga, Al Demers, Robbert Van Renesse, "Kelips: Building an Efficient and Stable P2P DHT through Increased Memory and Background Overhead," *In 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, pp.160-169, 2003.

[28] John Byers, Jeffrey Considine, Michael Mitzenmacher, "Simple Load Balancing for Distributed Hash Tables," *In 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, pp.80-87, 2003.

[29] Ananth Rao, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, Ion Stoica, "Load Balancing in Structured P2P Systems," *In 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, pp.68-79, 2003.