A READ-ONLY DISTRIBUTED HASH TABLE

VERDI MARCH

B.Sc (Hons) in Computer Science, University of Indonesia

A thesis submitted

for the degree of Doctor of Philosophy

Department of Computer Science

National University of Singapore

2007

DECLARATION

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Abstract

Distributed hash table (DHT) is an infrastructure to support resource discovery in large distributed system. In DHT, data items are distributed across an overlay network based on a hash function. This leads to two major issues. Firstly, to preserve ownership of data items, commercial applications may not allow a node to proactively store its data items on other nodes. Secondly, data-item distribution requires all nodes in a DHT overlay to be publicly writable, but some nodes do not permit the sharing of its storage to external parties due to a different economical interest. In this thesis, we present a DHT-based resource discovery scheme without distributing data items called R-DHT (***R***ead-only ***DHT***). We further extend R-DHT to support multi-attribute queries with our Midas scheme (***M***ult***i***-***d***imensional r***a***nge querie***s***).

R-DHT is a new DHT abstraction that does not distribute data items across an overlay network. To map each data item (e.g. a resource, an index to a resource, or resource metadata) back onto its resource owner (i.e. physical host), we virtualize each host into virtual nodes. These nodes are further organized as a segment-based overlay network with each segment consisting of resources of the same type. The segment-based overlay also increases R-DHT resiliency to node failures. Compared to conventional DHT, R-DHT's overlay has a higher number of nodes which increases lookup path length and maintenance overhead. To reduce

R-DHT lookup path length, we propose various optimizations, namely routing by segments and shared finger tables. To reduce the maintenance overhead of overlay networks, we propose a hierarchical R-DHT which organizes nodes as a two-level overlay network. The top-level overlay is indexed based on resource types and constitutes the entry points for resource owners at second-level overlays.

Midas is a scheme to support multi-attribute queries on R-DHT based on *d*-to-*one* mapping. A multi-attribute resource is indexed by a one-dimensional key which is derived by applying a Hilbert space-filling curve (SFC) to the type of the resource. The resource is then mapped (i.e. virtualized) onto an R-DHT node. To retrive query results, a multi-attribute query is transformed into a number of exact queries using Hilbert SFC. These exact queries are further processed using R-DHT lookups. To reduce the number of lookup required, we propose two optimizations to Midas query engine, namely incremental search and search-key elimination.

We evaluate R-DHT and Midas through analytical and simulation analysis. Our main findings are as follows. Firstly, the lookup path length of each R-DHT lookup operation is indeed independent of the number of virtual nodes. This demonstrates that our lookup optimization techniques are applicable to other DHT-based systems that also virtualize physical hosts into nodes. Secondly, we found that R-DHT is effective in supporting multi-attribute range queries when the number of query results is small. Our results also imply that a selective data-item distribution scheme would reduce cost of query processing in R-DHT. Thirdly, by not distributing data items, DHT is more resilient to node failures. In addition, data update at source are done locally and thus, data-item inconsistency is avoided. Overall, R-DHT is effective and efficient for resource indexing and discovery in large distributed systems with a strong commercial requirement in the ownership of data items and resource usage.

Acknowledgements

I thank God almighty who works mysteriously and amazingly to make things happen. I have never had the slightest imagination to pursue a doctoral study, and yet, His guidance has made me come this far. Throughout these five years, I also slowly learn to appreciate His constants blessings and love.

To my supervisor, A/P Teo Yong Meng, I express my sincere gratitude for his advise and guidance throughout my doctoral study. His determined support when I felt my research was going nowhere is truly inspirational. I learned from him the importance of defining research problems, how to put solutions and findings into perspective, a mind set of always looking for both sides of a coin, and technical writing skill. I also like to express my gratitude to my Ph.D. thesis committee, Professors Gary Tan Soon Huat, Wong Weng Fai, and Chan Mun Choon.

I acknowledge the contributions of Dr Wang Xianbing to this thesis. Due to his persistance, we managed to analytically prove the lookup path length of R-DHT. In addition, the backup-fingers scheme was invented when we discussed experimental results that are in contrast to theoretical analysis. I am indebted to Peter Eriksson (KTH, Sweden) who implemented a simulator that I use in Chapter 3. Dr Bhakti Satyabudhi Stephan Onggo (LUMS, UK) has provided me his advice regarding simulations and my thesis writing. Hendra Setiawan gave

me a crash course on probability theories to help me in performing theoretical analysis. Professor Seif Haridi (KTH, Sweden), Dr Ali Ghodsi (KTH, Sweden), and Gabriel Ghinita provided valuable inputs at various stages of my research. With Dr Lim Hock Beng, I have had some very insightful discussions regarding my research. I owe a great deal to Tan Wee Yeh, the keeper of Angsana and Tembusu2 clusters, whom I bugged frequently during my experiments. I thank Johan Prawira Gozali for sharing with me major works in job scheduling when I was looking for a research topic. Many thanks to Arief Yudhanto, Djulian Lin, Fendi Ciuputra Korsen, Gunardi Endro, Hendri Sumilo Santoso, Kong Ming Siem, and other friends as well for their support.

Finally, I thank my parents who have devoted their greatest support and encouragement throughout my tough years in NUS. I would never have completed this thesis without their constant encouragement especially when my motivation was at its lowest point. Thank you very much for your caring support.

Contents

List of Symbols

R-DHT

β	Ratio of the number of collisions in hierarchical R-DHT with *detect & resolve* to the number of collisions in hierarchical R-DHT *without detect & resolve*
ξ	Stabilization degree of an overlay network
ξ_n	Correctness of n's finger table
f	Finger
h	Host
K	Number of unique keys in a system
k	Key
N	Number of hosts
n	Node
p	Stabilization period
r	Resource
S_k	Segment prefixed with k
T	Average number of unique keys in a host
T_h	Set of unique keys in host h
V	Number of nodes

Midas

a	Length parameter that determines the size of query region for the experiments in Chapter 4
C	Number of clusters in query region

c	Cluster is consecutive Hilbert identifiers from $c.lo$–$c.hi$
d	Number of Dimensions
$f_{Hilbert}^{-1}$	Function to map a Hilbert identifier to a coordinate
$f_{Hilbert}$	Function to map a coordinate to a Hilbert identifier
H_l^d	The l^{th}-order Hilbert curve of a d-dimensional space
I	Number of intermediate nodes required to locate a responsible node
l	Approximation level of a multidimensional space and a Hilbert curve
Q	Query region whose $Q.lo$ and $Q.hi$ are its smallest and largest coordinates
q	Ordered set of search keys
Q_{akey}	Number of available keys
Q_{cnode}	Number of Chord nodes responsible for keys
Q_{skey}	Number of search keys
R	Number of responsible nodes

List of Figures

List of Tables

List of Theorems

Chapter 1

Introduction

The advance of internetworking has lead to initiatives to achieve the sharing and collaboration of resources across geographically dispersed locations. One popular initiative is peer-to-peer-based systems. Peer-to-peer (P2P) is an architecture for building large distributed systems that facilitate resource sharing among *nodes* (peers) from different administrative domains, where nodes are organized as an *overlay network* on top of existing network infrastructure (e.g. the TCP/IP network). The main characteristics of P2P are (i) every node can be a resource provider (server) and a resource consumer (client), and (ii) the overlay network are self-organizing with minimum manual configuration [10, 18, 100, 112].

P2P has been specifically applied for file-sharing applications [6]. However, the popularity of P2P paradigm has lead to its adoption by other types of applications such as information retrieval [105, 109, 127, 135, 146], filesystems [38, 39, 42, 46, 66, 81, 83, 104], database [70, 111], content delivery [34, 41, 48, 73, 82, 88, 125], and communication and messaging systems [3, 11, 12, 13, 102]. Recently, P2P has also been proposed to support resource discovery in computational grid [27, 28,

71, 91, 132, 145].

A key service in P2P is an effective and efficient resource discovery service. Effective means users should successfully find available resources with high *result guarantee*, while efficient means resource discovery processes are subjected to performance constraints such as minimum number of hops or minimum network traffic. As a P2P system is comprised of peer nodes from different administrative domains, an important design consideration of a resource discovery scheme is to address the problem of resource ownership and conflicting self-interest among administrative domains.

In this thesis, we present a resource discovery scheme based on *read-only* DHT (R-DHT). The remainder of this chapter is organized as follows. First, we review existing P2P lookup schemes in Section 1.1 and introduce a class of decentralized P2P lookup schemes called DHT in Section 1.2. In Section 1.3, we discuss how DHT supports a type of complex queries called multi-attribute range queries. Then, we highlight the problem of *data-item distribution* in Section 1.4. Next, we present the objective of this thesis and our contributions in Section 1.5–1.6. Finally, we describe the organization of this thesis in Section 1.7.

1.1 P2P Lookup

Based on the architecture, we classify P2P lookup schemes as *centralized* and *decentralized* (Figure 1.1).

Centralized schemes such as Napster [8] employ a directory server to index all resources in the overlay network. This leads to high result guarantee and efficiency since each lookup is forwarded only to the directory server. However, for large systems, a central authority needs a significant investment in providing a powerful

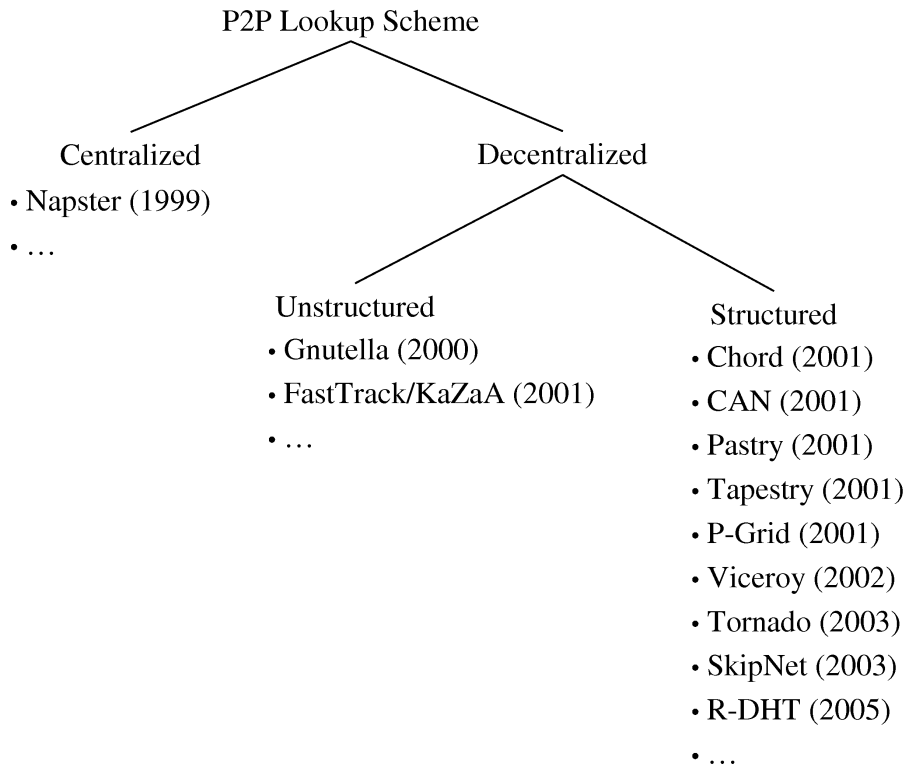


Figure 1.1: Classification of P2P Lookup Schemes

directory server to handle a high number of requests. The directory server is also a potential single point of failure due to technical reasons such as hardware failure, and non-technical reasons such as political or legal actions. A well-publicized example is the termination of Napster service in July 2001 due to legal actions.

Decentralized schemes minimize the reliance on a central entity by distributing the lookup processing among nodes in the overlay. Based on the overlay topology, decentralized schemes are further classified as *unstructured* P2P and *structured* P2P.

Unstructured P2P such as Gnutella [6] organizes nodes as a random overlay graph. In the earlier unstructured P2P, each node indexes only its own resources and a lookup floods the overlay: each node forwards an incoming lookup to all its neighbors. However, flooding limits scalability because in a P2P system consisting of

N nodes, the lookup complexity, in terms of the number of messages, is $O(N^2)$ [98, 121]. Hence, a high volume of network traffic is generated. To address this scalability issue, various approaches to limit search scope are proposed, including heuristic-based routing [15, 37, 79, 94, 141], distributed index [33, 35, 40], superpeer architecture [142], and clustering of peers [33, 114]. Though improving lookup scalability, limiting search scope leads to a lower result guarantee: a lookup returns a false negative answer when it is terminated before successfully locating resources. Thus, trying to efficiently achieve a high result guarantee remains a challenging problem [35, 138].

Structured P2P, also known as *distributed hash table* (DHT) [62, 69, 89, 117], is another decentralized lookup scheme that aims to provide a scalable lookup service with high result guarantee. We review the mechanism of DHT in Section 1.2 and how DHT supports complex queries in Section 1.3.

1.2 Distributed Hash Table (DHT)

DHT, as with a hash-table data structure, provides an interface to retrieve a key-value pair. A *key* is an identifier assigned to a resource; traditionally this key is a hash value associated with the resource. A *value* is an object to be stored into DHT; this could be the shared resource itself (e.g. a file), an index (pointer) to a resource, or a resource metadata. An example of a key-value pair is ⟨`SHA1(file_name)`, `http://peer-id/file`⟩, where the key is the SHA1 hash of the file name and the value is the address (location) of the file. DHT works in a similar way as hash tables. Whereas a hash table assigns every key-value pair onto a bucket, DHT assigns every key-value pair onto a node.

There are three main concepts in DHT: *key-to-node mapping*, *data-item distribution*, and *structured overlay networks*.

Key-to-Node Mapping Assuming that keys and nodes share the same identifier space, DHT maps key k to node n where n is the *closest* node to k in the identifier space; we refer to n as the *responsible node* of k. We use the term *one-dimensional DHT* and *d-dimensional DHT* to refer to DHT that use a one-dimensional identifier space and a d-dimensional identifier space, respectively.

Data-Item Distribution All key-value pairs (i.e. *data items*) whose key equals to k are *stored* at node n regardless of who owns these key-value pairs. To improve the resilience of lookups when the responsible node fails, the key-value pairs can also be replicated in a number of neighbors of n. However, the replication needs to consider application-specific requirements such as consistency among replicas, degree of replication, and overhead of replication [42, 54, 87, 113, 120].

Structured Overlay Network In DHT, nodes are organized as a structured overlay network with the purpose of striking a balance between routing performance and overhead of maintaining routing states. There are two important characteristics of a structured overlay network:

1. *Topology*

   A structured overlay network resembles a graph with a certain topology such as a ring [123, 133], a torus [116], or a tree [14, 99].

2. *Ordering of nodes*

   The position of a node in a structured overlay network is determined by the node identifier.

Compared to unstructured P2P, DHT is perceived to offer a better lookup performance in terms of results guarantee and lookup path length [93]. Due to the key-to-node mapping, finding a key-value pair equals to locating a node respon-

sible for the key. This increases result guarantee (i.e. a lower number of false negative answers) because it avoids the termination of lookups before existing keys are found[1]. By exploiting its structured overlay, DHT locates the responsible node in a shorter and bounded number of hops (i.e. the lookup path length).

Existing DHT implementations adopt all the three DHT main concepts. Two of these concepts, i.e. key-to-node mapping and structured overlay network, can be implemented differently among DHT implementations. On the other hand, data-item distribution is implemented in existing DHT by providing a *store* operation [43, 120]. As an illustration of how DHT concepts are implemented, we present three well-known DHT examples, namely Chord [133], Content-Addressable Network (CAN) [116], and Kademlia [99].

1. Chord, a one-dimensional DHT, is the basis for implementing our proposed read-only DHT scheme in Chapter 2–4.

2. CAN, a d-dimensional DHT, is used in an alternative implementation of our proposed scheme in Appendix A.

3. Kademlia is another one-dimensional DHT with a different key-to-node mapping function and structured overlay topology compared to Chord.

For each of these examples, we first elaborate on its overlay topology and key-to-node mapping function. We also highlight that each of the presented example distributes data items. Lastly, we discuss the process of looking up for a key (i.e. the basic DHT lookup operation) and the construction of overlay network.

[1]In contrast to DHT, the result guarantee in unstructured P2P depends on the popularity of key-value pairs. Lookup for popular key-value pairs, i.e. highly replicated and frequently requested, have a higher probability to return a correct answer compared to lookup for less popular key-value pairs [93].

1.2.1 Chord

Chord is a DHT implementation that supports $O(\log N)$-hops lookup path length and $O(\log N)$ routing states per node, where N denotes the total number of nodes [133] . Chord organizes nodes as a ring that represents an m-bit one-dimensional circular identifier space, and as a consequence, all arithmetic are modulo 2^m. To form a ring overlay, each node n maintains two pointers to its immediate neighbors (Figure 1.2). The successor pointer points to $successor(n)$, i.e. the immediate neighbor of n clockwise. Similarly, the predecessor pointer points to $predecessor(n)$, the immediate neighbor of n counter clockwise.

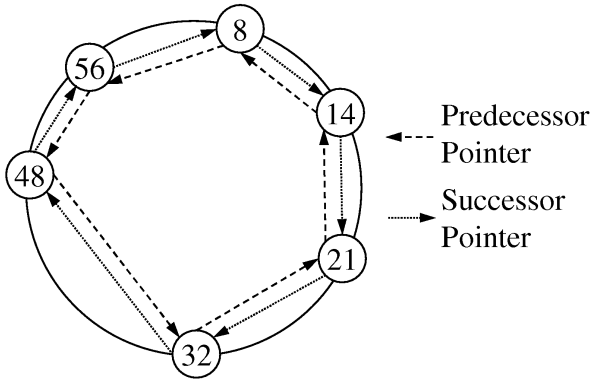
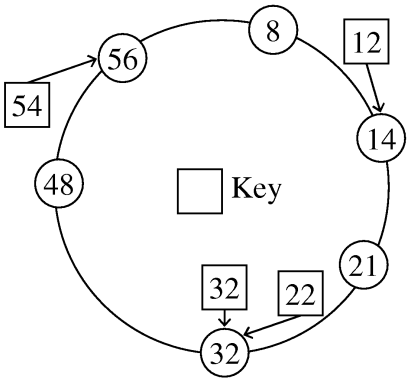


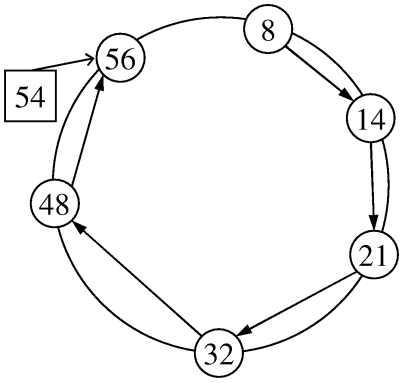
Figure 1.2: Chord Ring

Chord maps key k to $successor(k)$, the first node whose identifier is equal to or greater than k in the identifier space (Figure 1.3a). Thus, node n is responsible for keys in the range of $(predecessor(n), n]$, i.e. keys that are greater than $predecessor(n)$ but smaller than or equal than n. For example, node 32 is responsible for all keys in (21, 32]. All key-value pairs whose key equals to k are then stored on $successor(k)$ regardless of who owns the key-value pairs (i.e. data-item distribution).

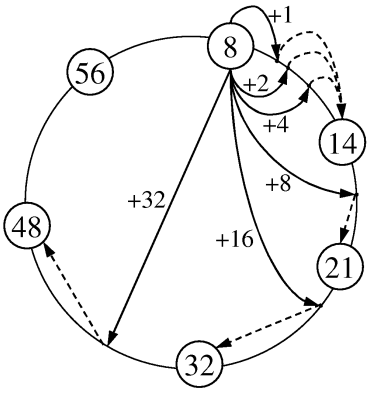
Finding key k implies that we route a request to $successor(k)$. The simplest approach for this operation, as illustrated in Figure 1.3b, is to propagate a re-



(a) Map and Distribute Keys to Nodes



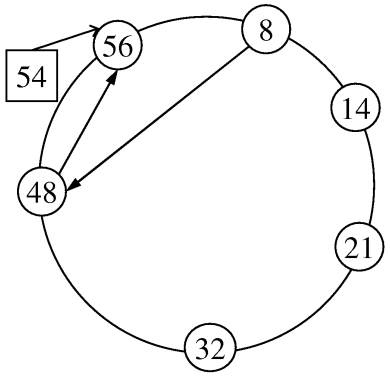
(b) Traverse the Ring to Find *successor*(54)



Finger Table of Node 8

8 + *1*	14
8 + *2*	14
8 + *4*	14
8 + *8*	21
8 + *16*	32
8 + *32*	48

(c) The Fingers of Node 8



(d) *find_successor*(*54*) Utilizing Finger Tables

Figure 1.3: Chord Lookup

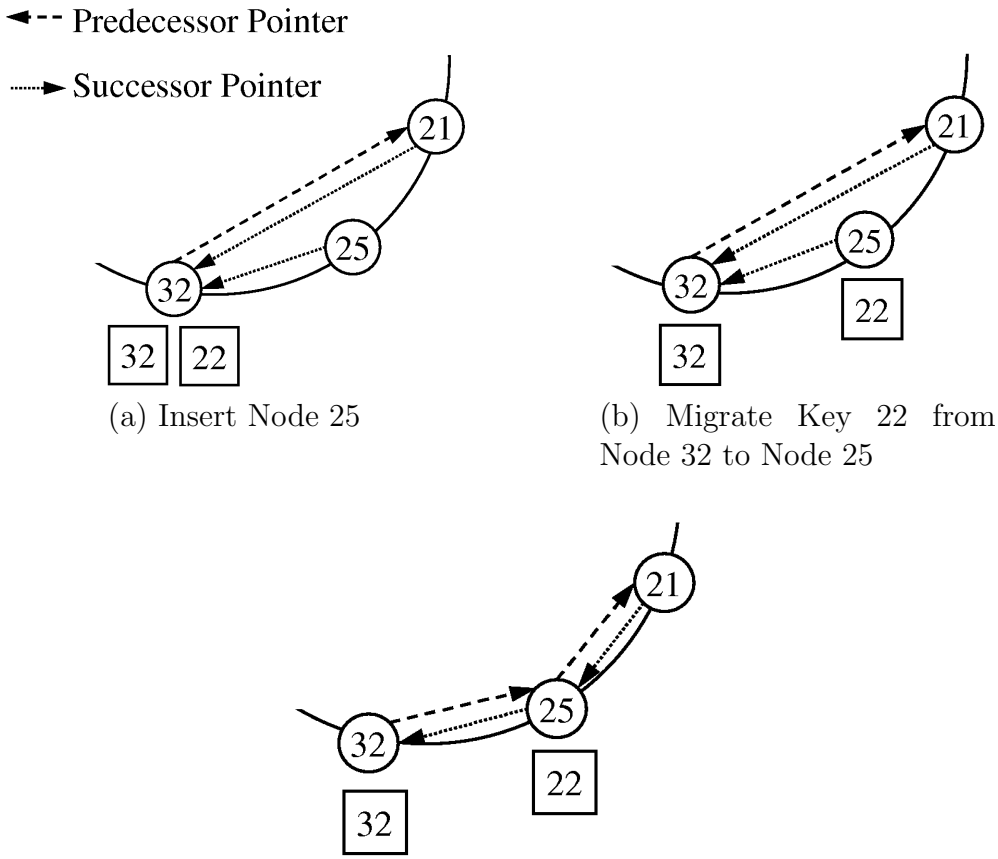
quest along the Chord ring in a clockwise direction until the request arrives at $successor(k)$. However, this approach is not scalable as its complexity is $O(N)$, where N denotes the number of nodes in the ring [133].

To speed-up the process of finding $successor(k)$, each node n maintains a finger table of m entries (Figure 1.3c). Each entry in the finger table is also called a finger. The i^{th} finger of n is denoted as $n.finger[i]$ and points to $successor(n + 2^{i-1})$, where $1 \leq i \leq m$. Note that the 1^{st} finger is also the successor pointer while the largest finger divides the circular identifier space into two halves. When $N < 2^m$, the finger table consists of only $O(\log N)$ unique entries.

By utilizing finger tables, Chord locates $successor(k)$ in $O(\log N)$ hops with high probability [133]. Intuitively, the process resembles a binary search where each step halves the distance to $successor(k)$. Each node n forwards a request to the nearest known preceding node of k. This is repeated until the request arrives at $predecessor(k)$, the node whose identifier precedes k, which will forward the request to $successor(k)$. Figure 1.3d shows an example of finding $successor(54)$ initiated by node 8. Node 8 forwards the request to its 6^{th} finger which points to node 48. Node 48 is the predecessor of key 54 because its 1^{st} finger points to node 56 and $48 < 54 \leq 56$. Thus, node 48 will forward the request to node 56.

Figure 1.4 illustrates the construction of a Chord ring. A new node n joins a Chord ring by locating its own successor. Then, n inserts itself between $successor(n)$ and the predecessor of $successor(n)$, illustrated in Figure 1.4a. The key-value pairs stored on $successor(n)$, whose key is less than or equal to n, is migrated to node n (Figure 1.4b). Because the join operation invalidates the ring overlay, every node performs periodic stabilizations to correct its successor and predecessor pointers (Figure 1.4c), and its fingers.

finger-correction mechanism to correct its successor and predecessor pointers (Figure 1.4c).



(a) Insert Node 25

(b) Migrate Key 22 from Node 32 to Node 25

(c) Correct Successor and Predecessor Pointers

Figure 1.4: Join Operation in Chord

1.2.2 Content-Addressable Network

CAN is d-dimensional DHT that supports $O(n^{1/d})$-hops lookup path length and $O(d)$ routing states per node, where N denotes the total number of nodes [116]. The design of CAN is based on a d-dimensional Cartesian coordinate space on a d-torus. The coordinate space is partitioned into zones and every node is responsible for a zone. Each node is also assigned a virtual identifier (VID) that reflects its position in the coordinate space. To facilitate routing (i.e. lookups) , a node maintains pointers to its adjacent neighbors. For a d-dimensional coordinate space

partitioned into N equal zones, every node maintains $2d$ neighbors. Figure 1.5 illustrates an example of 2-dimensional CAN consisting of six nodes and an 8×8 coordinate space. Node E, whose VID is 101, is responsible for zone [6–8, 0–4] where the lower-left Cartesian point (6, 0) and the upper-right Cartesian point (8, 4) are the lowest and highest coordinates in this zone, respectively.

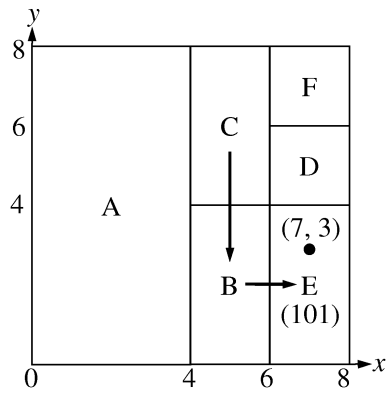


Figure 1.5: Lookup in a 2-Dimensional CAN

CAN maps key k to point p within a zone. As in Chord, CAN also adopts data-item distribution where the key-value pair whose key equals to k is stored to the node responsible for the zone. Thus, finding a key implies locating the zone that contains point p. Intuitively, CAN routes a request to a destination zone by using a straight line path from the source to the destination. Each node forwards a request to its neighbor whose coordinate is the closest to the destination coordinate. For a d-dimensional coordinate space divided into N equal zones, the lookup path length is $O(n^{1/d})$ [116]. Figure 1.5 shows a lookup for a key mapped to Cartesian point (7, 3). Initiated by node C, the lookup is routed to node E as its zone, [6–8, 0–4], contains the requested point.

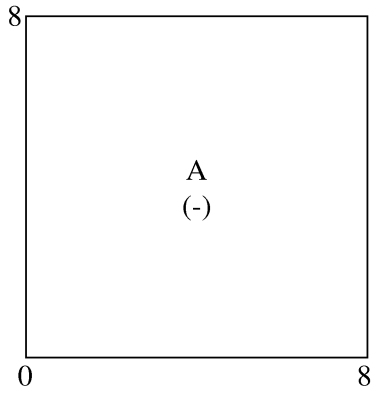
To join a CAN coordinate space, a new node n randomly chooses a point p and locates zone z that contains p. Then, z is split into two child zones along a particular dimension based on a well-defined ordering. For instance, in a two-

dimensional CAN, a zone is first split along the x axis followed by the y axis. Node e, which was responsible for z, will take over the lower child zone along the split dimension, while the new node n is responsible for the higher child zone. To properly reflect their new position, the VIDs of both nodes are updated by concatenating the original VID of e with 0 (if the node in the lower child zone) or 1 (if the node is in the higher child zone).

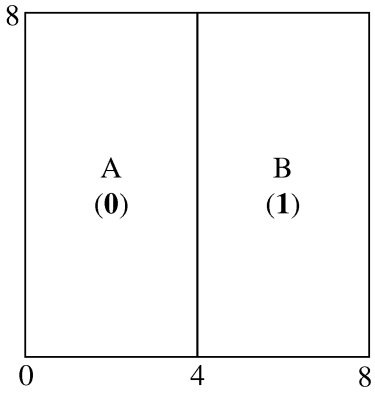
Figure 1.6 illustrates the construction of a 2-dimensional CAN consisting of six nodes. A binary string in parentheses denotes a node VID. Initially, the first node A is responsible for the whole coordinate space, i.e. [0–8, 0–8], and its VID is an empty-string (Figure 1.6a). As node B arrives (Figure 1.6b), zone [0–8, 0–8] is split along the x axis into two child zones: [0–4, 0–8] and [4–8, 0–8], which corresponds to the lower and higher zone, respectively, along the x axis. Node A will be responsible for the lower child zone and therefore, its new VID is **0**, which is the concatenation of A's original VID and **0**. Meanwhile, the new node B is responsible for the higher child zone and its VID will be **1**. Figure 1.6c shows another node C arrives and further splits zone [4–8, 0–8]. Because zone [4–8, 0–8] is the result of a previous splitting along the x axis, this zone is now split along the y axis, which results in [4–8, 0–4], i.e. the lower child zone along the y axis, and [4–8, 4–8], i.e. the higher child zone along the y axis. Node B will be taking over the lower child zone and its new VID will be 1**0**. The new node C is responsible for the higher child zone and therefore, its VID will be 1**1**. The zone splitting continues as more nodes join (Figure 1.6d–1.6f).

1.2.3 Kademlia

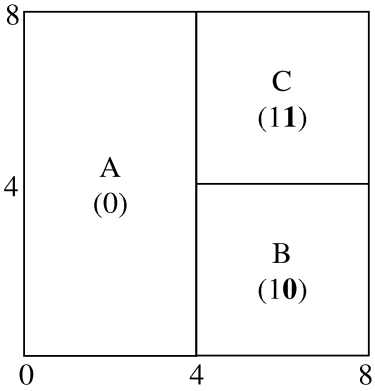
Assuming an m-bit identifier space, Kademlia supports $O(\log N)$-hops lookup path length and $O(\kappa m)$ routing states per node, where N denotes the total number of nodes and κ denotes a coefficient for routing-states redundancy [99]. Kademlia



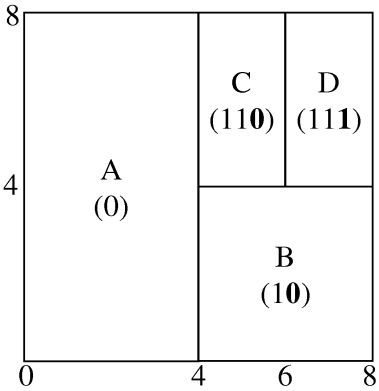
(a) Node A Occupies [0–8, 0–8]



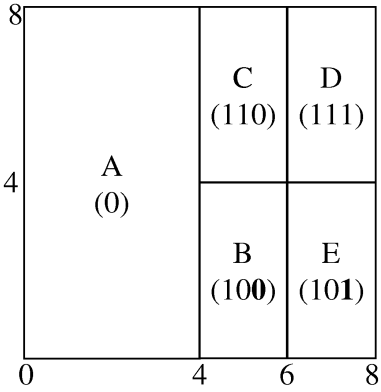
(b) Node B Splits [0–8, 0–8] along x Axis into [0–4, 0–8] and [4–8, 0–8]



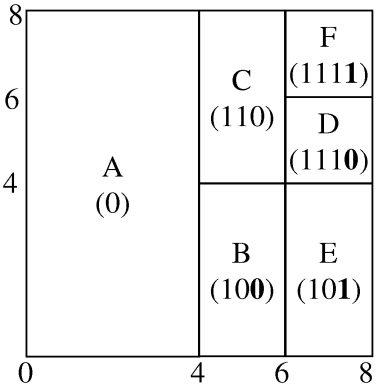
(c) Node C Splits [4–8, 0–8] along y Axis into [4–8, 0–4] and [4–8, 4–8]



(d) Node D Splits [4–8, 4–8] along x Axis into [4–6, 4–8] and [6–8, 4–8]



(e) Node E Splits [4–8, 0–4] along x Axis into [4–6, 0–4] and [6–8, 0–4]



(f) Node F Splits [6–8, 4–8] along y Axis into [6–8, 4–6] and [6–8, 6–8]

Figure 1.6: Dynamic Partitioning of a 2-Dimensional CAN

organizes nodes as a prefix-based binary tree where each node is a leaf of the tree. The position of a node is determined by the shortest unique prefix of the node identifier. Figure 1.7 illustrates the position of node 5 (0101_2) in a Kademlia tree, assuming a 4-bit identifier space.

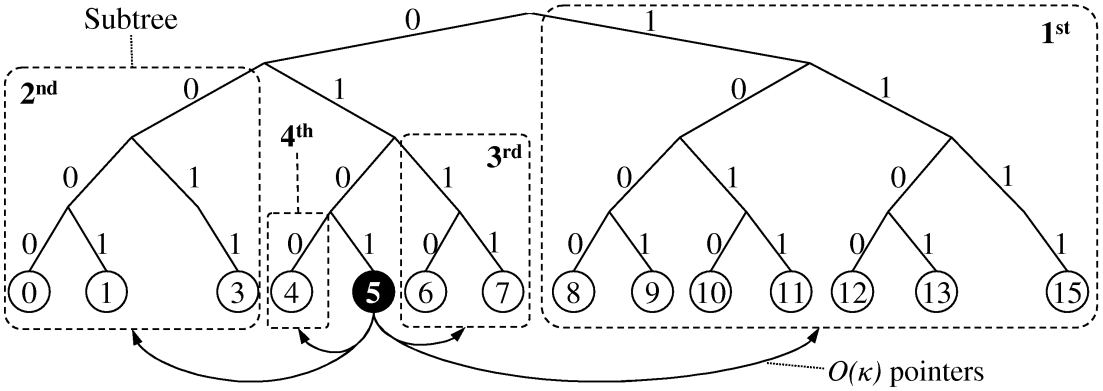


Figure 1.7: Kademlia Tree Consisting of 14 Nodes ($m = 4$ Bits)

To facilitate the routing of lookup requests, each node maintains a routing table consisting of $O(m)$ buckets where each bucket consists of $O(\kappa)$ pointers. First, node n divides the tree into m subtrees such that the i^{th} subtree consists of $O(N/2^i)$ nodes with the same $(i-1)$-bit prefix as n, where $1 \leq i \leq m$ and N denotes the number of nodes. The i^{th} subtree is *higher* than the j^{th} subtree if $i < j$. Thus, the 1^{st} subtree is also called the *highest* subtree, while the m^{th} subtree is the *lowest* subtree. For each subtree, node n maintains a bucket consisting of pointers to $O(\kappa)$ nodes in the subtree. Figure 1.7 illustrates the routing states maintained by node 5. The node partitions the binary tree into four subtrees. The 1^{st} subtree consists of nodes with prefix 1, which amount to (nearly) half of the tree. The remaining three subtrees consists of nodes with prefix 0, 01, and 010, respectively.

Kademlia maps key k to node n whose identifier is the closest to k. The distance between k and n is defined as $d(k, n) = k \oplus n$ where $\oplus$ is an XOR operator and the value of $d(k, n)$ is interpreted as an integer. Then, key-value pairs whose key

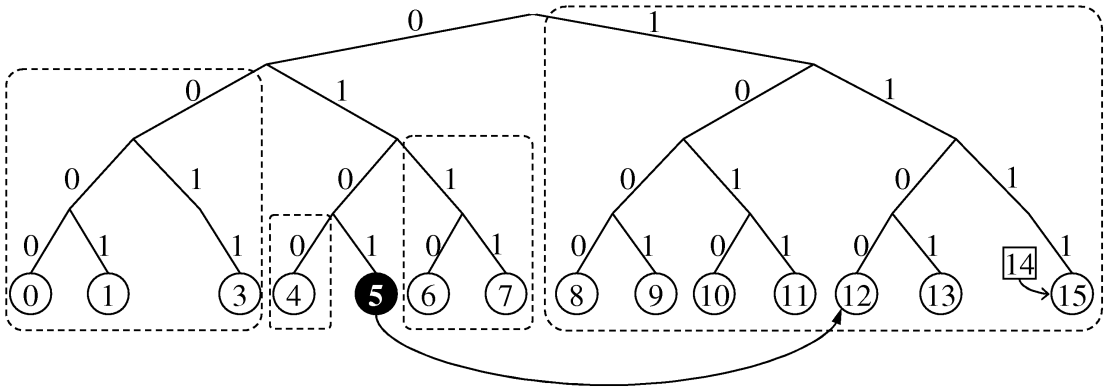
equals to k are distributed to n. To find key k, each node forwards a lookup request to the lowest subtree that contains k, i.e. a subtree that has the same longest common prefix as k. This is repeated until the request arrives at the node closest to k. In an N-nodes tree, the lookup complexity is $O(\log N)$ hops and the reason is similar to Chord: every routing step halves the distance to the destination. Kademlia reduces the turnaround time of lookups by exploiting its κ-bucket routing tables. When forwarding a request to a subtree, the request is concurrently send to α ($\leq \kappa$) nodes in the subtree.

Figure 1.8a illustrates a lookup for key 14 (1110_2) initiated by node 5 (0101_2). The key is mapped to node 15 where $d(14, 15) = 1$ (0001_2). Because key 14 and node 5 do not share a common prefix, node 5 forwards the request to any node in the 1^{st} subtree (Figure 1.8a). Assuming that the request arrives at node 12 (1100_2), node 12 further forwards the request to its 3^{rd} subtree which contains only node 15 (Figure 1.8b). At node 15 (1111_2), the lookup request will be terminated because the distance between k and any node in node 15's lowest subtrees is larger than $d(14, 15)$ (Figure 1.8c).

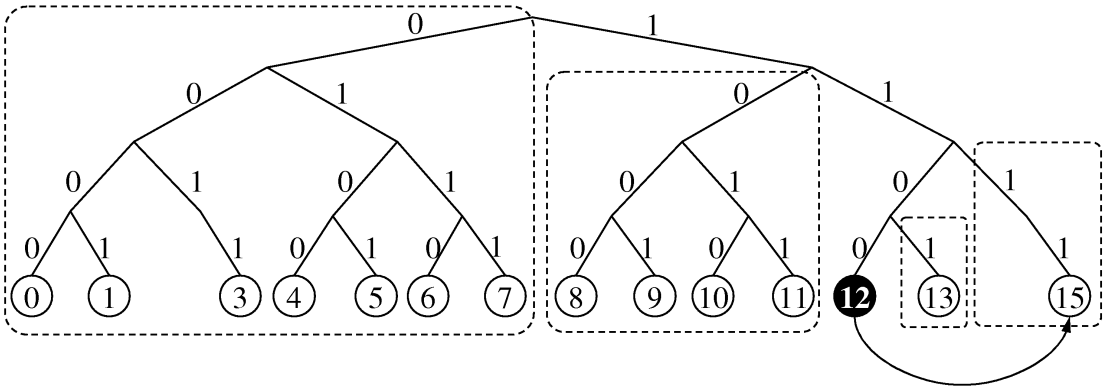
The construction of a Kademlia tree is straightforward. A new node n first locates another node n' closest to it. Then, n probes and builds its m subtrees through node n'. In addition, every time n receives a request, it adds the sender of the request into the appropriate bucket. The replacement policy will ensure that a bucket contains pointers to stable nodes (i.e. nodes with longer uptime).

1.3 Multi-Attribute Range Queries on DHT

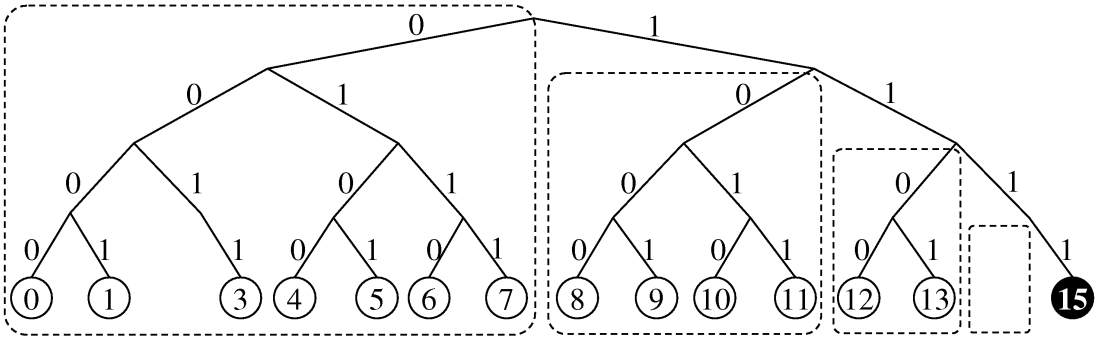
The DHT lookup operation, presented in the previous section, offers high results guarantee and short lookup path length for *single-attribute exact queries* [93]. This may suffice the needs of some applications such as CFS [42] and POST [102].



(a) Node 5 Initiates a Lookup for Key 14 (1110_2)



(b) Node 12 Processes the Lookup



(c) Node 15 Terminates the Lookup

Figure 1.8: Kademlia Lookup ($\alpha = 1$ Node)

However, applications such as computational grid deal with resources described by many attributes [5, 7]. Users of such applications needs to find resources that match a *multi-attribute range query*. To fulfill the need of such applications, DHT must support not only single-attribute exact queries (i.e. the basic DHT lookup operation), but also multi-attribute range queries.

A multi-attribute range query is a query that consist of multiple search attributes. Each search attribute can be constrained by a range of values using relational operators $<$, $\leq$, $=$, $>$, and $\leq$. An example of such queries is to *find compute resources whose* `cpu` $=$ *P3 and 1 GB* $\leq$ `memory` $\leq$ *2 GB*. A special case of multi-attribute range queries is multi-attribute exact queries where each attribute is equal to a specific value. An example of a multi-attribute exact query is to *find compute resources whose* `cpu` $=$ *P3 and* `memory` $=$ *1 GB*. Supporting multi-attribute range queries is very well researched in other fields such as database [49] and information retrieval [21]. This thesis focuses on multi-attribute range queries on DHT.

As illustrated in Figure 1.9, we classify multi-attribute range query processing on DHT into three categories, namely distributed inverted index, d-to-d mapping, and d-to-*one* mapping. Distributed inverted index and d-to-*one* mapping scheme are applicable to both one-dimensional DHT [99, 123, 133, 144] and d-dimensional DHT [116], whereas d-to-d mapping is applicable to d-dimensional DHT only. In Chapter 1.3.1–1.3.3, we discuss the indexing scheme and query-processing scheme used in each of the categories.

1.3.1 Distributed Inverted Index

For every resource that is described by d attributes, distributed inverted index assigns d keys to the resource, i.e. one key per attribute. To facilitate range queries, each attribute is hashed into a key using a locality-preserving hash func-

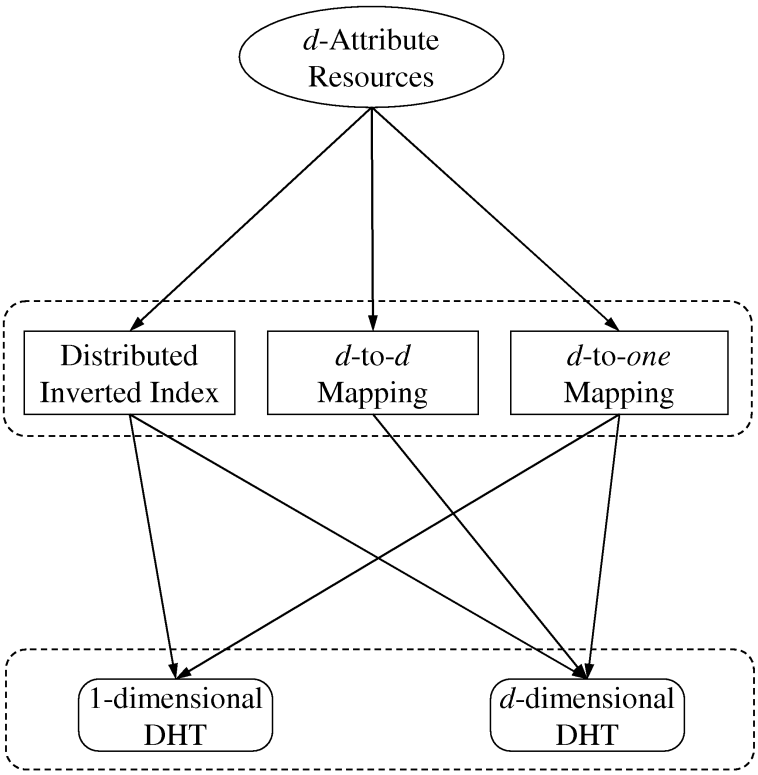


Figure 1.9: Classification of Multi-Attribute Range Query Schemes on DHT

tion [19, 28]; this ensures that consecutive attributes are hashed to consecutive keys. Examples of DHT-based distributed inverted index are MAAN [28], CANDy [24], n-Gram Indexing [67], KSS [56], and MLP [129]. Figure 1.10 illustrates the indexing of a compute resource R with two attributes, `cpu` = P3 and `memory` = 1 GB. Based on these attributes, we assign two key-value pairs to the resource, one with key $k_{cpu} = hash(P3)$ and the other with key $k_{memory} = hash(1GB)$. Then, we store the two key-value pairs to the underlying DHT.

There are two main strategies for processing a d-attribute range query. The first strategy uses $O(d)$ DHT lookups; one lookup (i.e. the selection operator, σ, in relational algebra) for each attribute. The result sets of these lookups need to be intersected (i.e. operator $\cap$) to produce a final result set. This can be performed at the query initiator [28] or by pipelining intermediate result sets through a number of nodes [24, 56, 129], as illustrated in Figure 1.11. The second strategy requires

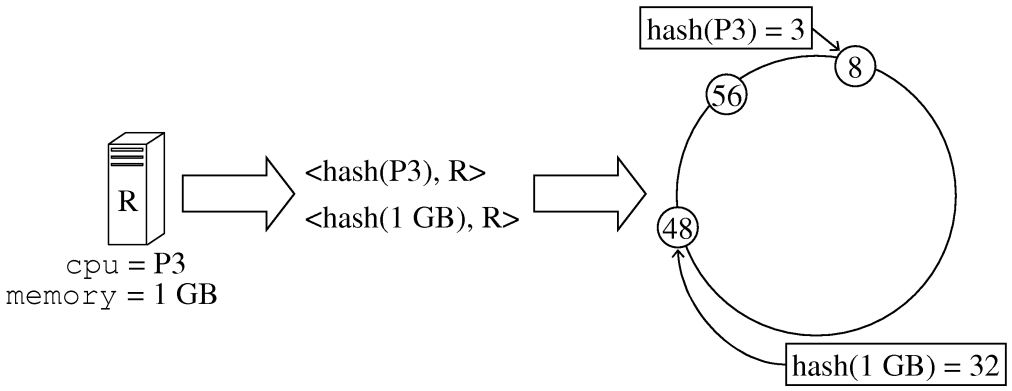


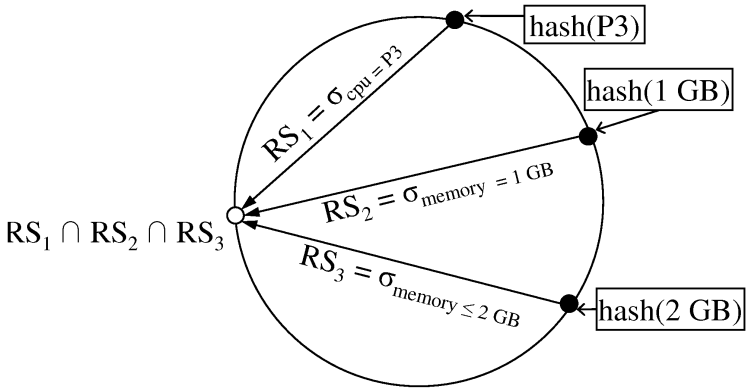
Figure 1.10: Example of Distributed Inverted Index on Chord

only $O(1)$ lookup to obtain the final result set. Assuming that each key-value pair also includes the complete attributes of the resource (value), the intersection can be performed only once.

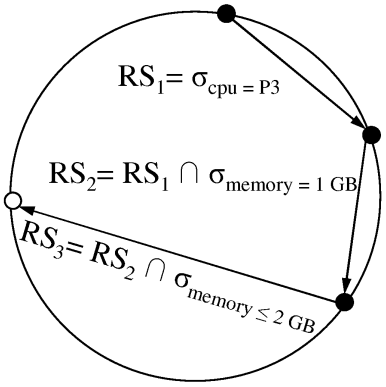
1.3.2 *d*-to-*d* Mapping

d-to-*d* mapping such as pSearch [135], MURK [50], and 2CAN [16], maps each *d*-attribute resource onto a point in a *d*-dimensional space. Figure 1.12 illustrates a compute resource with `cpu` = P3 and `memory` = 1 GB is mapped to point (P3, 1 GB) in a 2-dimensional CAN. The x-axis and y-axis of the coordinate space correspond to attribute `cpu` and `memory`, respectively.

In *d*-to-*d* mapping, a *d*-attribute range query can be visualized as a region in the coordinate space. For example, the shaded rectangle in Figure 1.12 represents a query for resources with any type of `cpu` and $256 \leq$ `memory` ≤ 768. The basic concept in processing a query involves two stages. First, a request is routed to any point in the query region. On reaching the initial point, the request is further flooded to the remaining points in the query region.



(a) At Query Initiator



(b) At Intermediate Nodes

Figure 1.11: Intersecting Intermediate Result Sets

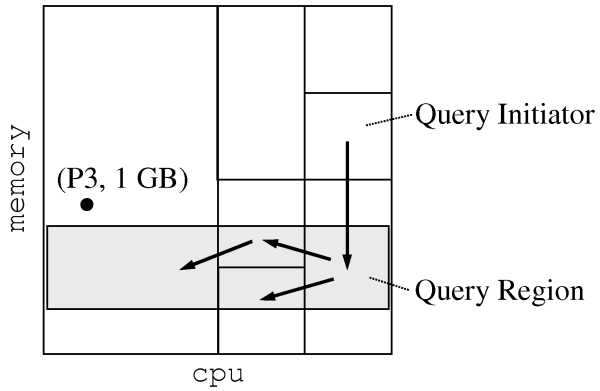


Figure 1.12: Example of Direct Mapping on 2-dimensional CAN

1.3.3 *d*-to-*one* Mapping

d-to-*one* mapping maps a *d*-attribute resource onto a point (i.e. a key) in a one-dimensional identifier space. Each *d*-attribute resource is assigned with a key

drawn from a one-dimensional identifier space. The key is derived by hashing the d-attribute resource using a locality-preserving function, i.e. the d-to-*one* mapping function. The resulted key (and key-value pair) is then stored on the underlying DHT. Compared to d-to-d mapping, d-to-*one* mapping can use one-dimensional DHT (e.g. Chord [133]) as the underlying DHT, as well as d-dimensional DHT (e.g. CAN [116]). Examples of query processing schemes on DHT that are based on d-to-*one* are Squid [127], SCRAP [50], ZNet [131], CISS [86], and CONE [16]. With the exception of CONE, all the above examples use space-filling curve (SFC) as the hash function. Figure 1.13 shows an example of Hilbert SFC [124] that maps each two-dimensional coordinate point onto an identifier, e.g. coordinate (3, 3) is mapped onto identifier 10.

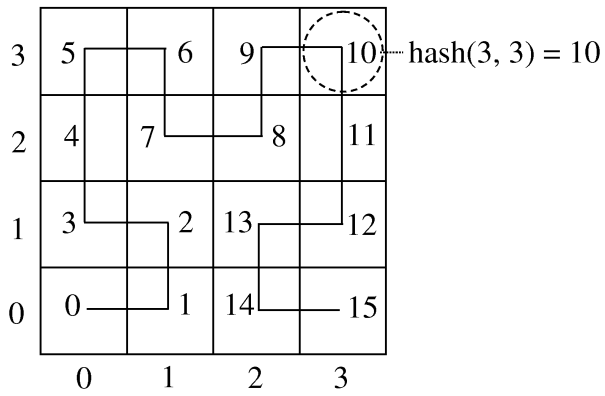
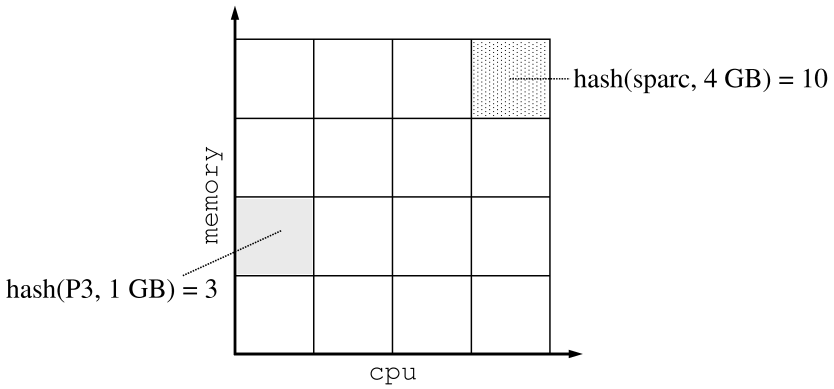


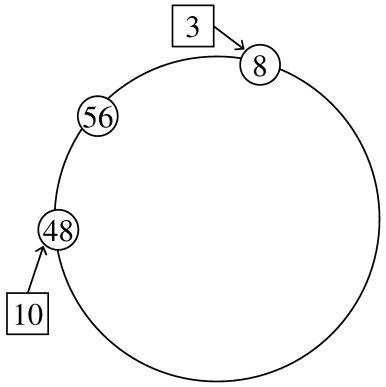
Figure 1.13: Hilbert SFC Maps Two-Dimensional Space onto One-Dimensional Space

Figure 1.14 illustrates the indexing of resources with two attributes. Each resource corresponds to a point in the 2-dimensional attribute space, and each point is further hashed into a key (Figure 1.14a). Using Hilbert curve, (`cpu` = P3, `memory` = 1 GB) and (`cpu` = sparc, `memory` = 4 GB) are assigned key 3 and key 10, respectively. Since each key is one-dimensional, it can be mapped directly to one-dimensional DHT such as Chord (Figure 1.14b).

Similar to d-to-d mapping, a d-attribute range query can be visualized as a re-



(a) Map Points in 2-Dimensional Attribute Space to Keys in 1-Dimensional Identifier Space



(b) Map Keys to Chord Nodes

Figure 1.14: Example of 2-Dimensional Hash on Chord

gion in the d-dimensional attribute space. However, the difference between d-to-d mapping and d-to-*one* mapping is in the query processing. In d-to-*one* mapping, we apply the d-to-*one* mapping function to the query region to produce a number of *search keys*. A naive way of searching is to issue a lookup for each search key. To reduce the number of lookups initiated, query processing is optimized by exploiting the facts that (i) some search keys do not represent available resources, and (ii) several search keys are mapped onto the same DHT node.

1.4 Motivation

Existing DHT distribute data items where key-value pairs are proactively distributed by their owner across the overlay network. As each DHT node stores its key-value pair (i.e. data item) to a *responsible node* which is determined by a key-to-node mapping function, data items from many nodes are aggregated in one responsible node. To exploit this property, various performance optimizations are proposed, including load balancing schemes [57, 58, 78], replication schemes to achieve high-availability [42, 54, 81, 83, 87], and data aggregation scheme to support multi-attribute range queries (see Section 1.3).

Though facilitating many performance optimizations in DHT, data-item distribution also reduces the autonomy (i.e. control) of nodes in placing their key-value pairs [44].

1. Node n has no control on where its key-value pairs will be stored because:
   (a) A key-to-node mapping function considers only the distance between keys and nodes in the identifier space.
   (b) A key can be remapped due to a new node as illustrated in Figure 1.4b.

   Hence, node n perceives its key-value pairs to be distributed to random nodes.

2. To join a DHT-based system, node n must make provision to store key-value pairs belonging to other nodes. However, n has limited control on the number of key-value pairs to store because:
   (a) The number of keys mapped to n is affected by n's neighbors (e.g. $predecessor(n)$ in Chord).
   (b) The number of key-value pairs with the same key (i.e. resources of the same type) depends on the popularity of the resource type; this is

beyond the control of n.

The limited node autonomy potentially hinders the widespread adoption of DHT by commercial entities. In large distributed systems, nodes can be managed by *different* administrative domains, e.g. different companies, different research institutes, etc. This has been observed in computational grid [47, 80] as well as earlier generations of distributed systems such as file-sharing P2P [6] and world wide web (WWW). In such applications, distributing data items among different administrative domains (in particular, different commercial entities) leads to two major issues:

Ownership of Data Items Commercial application requirements may not allow a node to proactively store its data items (even if data items are just pointers to a resource) on other nodes. Firstly, the node is required to ensure that it is the sole provider of its own data items. As an example, a web site may not allow its contents to be hosted or even directly linked by other web sites which include search engines, to prevent customers being drawn away from the originating web site [107, 108, 118]. Secondly, a node may restrict distributing its data items to prevent the misuse of its data items [55, 59, 60].

Though a node can encrypt its key-value pairs before storing them to other nodes, we argue that encryption addresses *privacy* issue instead of the ownership issue. The privacy issue is concerned with ensuring that data items are not accessible to illegitimate users and this is addressed by encrypting data items. On the other hand, in the case of ownership issue, data items are already publicly accessible.

Conflicting Self-Interest among Administrative Domains Data-item distribution requires all nodes in a DHT overlay to be publicly writable. However,

this may not happen when nodes do not permit the sharing of its storage resources to external parties due to a different economical interest. Firstly, nodes want to protect their investment in their storage infrastructure by not storing data items belonging to other nodes. Secondly, individual node may limit the amount of storage it offers. However, limiting the amount of storage reduces result guarantee if the total amount of storage in DHT becomes smaller than the total number of key-value pairs.

In addition to the problem in enforcing storage policies, nodes also face a challenge where their infrastructure is used by customers of other parties [110, 130]. As an example, when a node stores many data items belonging to other parties, the node experiences an increased usage of its network bandwidth and computing powers due to processing a high number of lookup requests for data items.

The above two issues can be addressed by not distributing data items. However, by design, DHT assumes that data items can be distributed across overlay networks.

1.5 Objective

User requirements may dictate P2P systems to provide an effective and efficient lookup service without distributing data items. In this thesis, we investigate a DHT-based approach without distributing data items and with supports for multi-attribute range queries. The proposed scheme consists of two main parts: R-DHT (***R***ead-only ***DHT***) and Midas (***M***ult***i***-***d***imensional r***a***nge querie***s***). R-DHT serves as the basic infrastructure to support the DHT lookup operations (i.e. single-attribute exact queries), and Midas adds supports for multi-attribute range queries on R-DHT. As an example, we apply our proposed scheme to support decentralized resource indexing and discovery in large computational grids [47, 80].

R-DHT is a class of distributed hash tables where a node allows "read-only" accesses to its key-value pairs, but does not allow key-value pairs belonging to other nodes to be written (mapped) on it. The design criteria for R-DHT include:

1. Support for DHT-style lookup.

   R-DHT must support the *flat naming scheme* in order to provide the hash table abstraction of locating data items. The lookup operation of R-DHT requires users to specify only the key of requested data items.

2. Effective and efficient lookup performance.

   The result guarantee and the lookup path length of R-DHT must not be worse than conventional DHT.

3. Lookup resiliency to node failures.

   R-DHT must be resilient to node failures even when resources by nature cannot be replicated. As an example, in a computational grid, resources are not replicable. However, there are multiple resource instances, shared by different administrative domains, with the same resource type, and finding a subset of these resources is sufficient. These properties are exploited to increase lookup resiliency in R-DHT.

   In this thesis, we do not focus on "availability" in which resources are replicated so that they can be located even if their master copy ceases to exist. Similar to DHT, resource replications can be introduced in R-DHT to increase resource availability.

Midas is a scheme to support multiple-attribute range queries on R-DHT. Midas indexes multi-attribute resources by mapping each of the resources onto an R-DHT node. In this thesis, we focus on resources whose description conforms to a well-defined schema such as GLUE schema [5]. Midas processes multi-attribute

range queries using one or more R-DHT lookup operations. The design criteria of Midas include:

1. Efficient resource indexing.

   To reduce the overhead of indexing resources, each multi-attribute resource is assigned only one key so that the resource is mapped onto one R-DHT node only. In addition, the indexing exploits *locality* where resources with similar attributes are mapped to R-DHT nodes that are close in the overlay network.

2. Efficient query processing.

   Midas processes a multi-attribute range queries by invoking one or more R-DHT lookup operations. The number of lookup operations and the number of intermediate hops per lookup must be minimized.

3. Support for one-dimensional overlay network.

   The majority of existing DHT implementations are one-dimensional DHT [99, 123, 133, 144]. Therefore, Midas must support one-dimensional R-DHT (e.g. Chord-based R-DHT) as the underlying DHT, in addition to d-dimensional R-DHT (e.g. CAN-based R-DHT).

1.6 Contributions

Our three main contributions are the R-DHT approach, hierarchical R-DHT, and multi-attribute range queries on R-DHT.

R-DHT Approach

Our proposed scheme enables DHT to map keys to nodes without distributing data items [97, 136]. The *read-only mapping scheme* in R-DHT virtualizes a *host* (i.e. a physical entity that shares resources) into *nodes*: one node is associated with

each unique key belonging to the host. Nodes are organized as a segment-based structured overlay network. The node identifier space is split into two sub-spaces: key space and host identifier space. Our scheme inherits the good properties of DHT presented in Chapter 1.2, namely support for decentralized lookups to minimize single point of failure, high result guarantee, and bounded lookup path length.

Compared to existing DHT, the R-DHT scheme results in the following benefits:

1. *Node autonomy in placing data items*

   In R-DHT, each node stores only its own key-value pairs, i.e. the index of its shared resources, without depending on a third-party publicly-writable nodes. Thus, nodes are read-only because they store only their own key-value pairs, as in the "pay-for-your-own" usage model [22]. Updates to a key-value pair are reflected immediately by the node that owns it. This avoids data inconsistency as in conventional DHT.

2. *Lookup performance*

   Though the size of R-DHT overlay is larger than DHT, the lookup path length in R-DHT is at worst equal to DHT. The segment-based overlay of R-DHT allows *messages to be routed by segments* and *finger tables to be shared among nodes*. When the number of unique resource types (K) is larger than the number of hosts (N), e.g. in file-sharing P2P systems, the lookup path length in R-DHT is bounded by $O(\log N)$ as in traditional DHT (Chord). However, when $K \leq N$, e.g. in computational grid, the $O(\log K)$-hops lookup path length of R-DHT is shorter than traditional DHT.

3. *Lookup resiliency to node failures*

   We demonstrate that R-DHT segment-based overlay reduces the number of failed lookups (i.e. lookups that return a false negative or false positive an-

swer) in the event of node failures. In R-DHT, lookups for key-value pairs shared by many nodes are more likely to succeed even without replicating key-value pairs. When one of the node fails, only its own key-value pairs become unavailable. The remaining key-value pairs in other nodes can still be discovered because R-DHT exploits segment-based overlay by using *backup fingers*. Thus, the probability to find resources of a certain type is higher when there are many nodes sharing resources of that type.

4. *Reuse of DHT functionalities*

   R-DHT reuses the existing functionalities from conventional DHT and as such, improvements in DHT are beneficial to R-DHT as well. To demonstrate this ability of R-DHT, we present R-Chord, an implementation of R-DHT scheme that uses Chord as the underlying overlay graph. R-Chord uses Chord's join algorithm to construct its overlay network. In addition, R-Chord's lookup and stabilization are based on Chord's lookup and stabilization algorithm.

We show the performance and overhead of R-DHT scheme through theoretical and simulation analysis.

Hierarchical R-DHT

We propose the design of a two-level R-DHT to reduce the maintenance overhead of R-DHT overlay networks. The hierarchical R-DHT partitions the maintenance overhead among smaller sub-overlays. To address the problem of collisions in the top-level overlay, we propose a scheme to detect and resolve collisions in hierarchical R-DHT [96]. Collisions occur in the top-level overlay because of membership changes when node joins or fails. We evaluate the effectiveness of this scheme through simulations.

Multi-Attribute Range Queries on R-DHT

Midas is our proposed scheme to support multi-attribute range queries on R-DHT [95]. The indexing scheme and query engine in Midas are based on *d*-to-*one* for the following reasons:

1. *One key per resource*

   d-to-*one* assigns one key to each resource so that the resource is later mapped onto one R-DHT node only. This reduces the overhead of indexing resources on R-DHT.

   Midas uses Hilbert space-filling curve [124] as the *d*-to-*one* mapping function because studies have indicated its effectiveness in preserving locality of multi-dimensional indexes [74, 103].

2. *Support for efficient query processing*

   Midas minimizes the number of lookup operations in processing a query by invoking lookups only for available resources. In addition, Midas exploits the locality of resource indexes to minimize the number of intermediate hops per lookup operation.

3. *Support for one-dimensional overlay network*

   Midas assigns to each resource a key which is drawn from a one-dimensional key space. Therefore, resources can be mapped onto a node in a one-dimensional R-DHT.

Using simulations, we show that query processing on R-DHT achieves a higher result guarantee than conventional DHT. We also study the implication of data-item distribution to the cost of processing queries. Our study indicates that for the same size of queries, the cost of query processing in conventional DHT and R-DHT is determined by the number of nodes and the number of query results,

respectively. This implies that R-DHT is more suitable when the number of query results is small. To reduce the query cost in R-DHT when the number of query results is large, an R-DHT-based system may perform data-item distribution only among a set of trusted nodes and search for query results only within the trusted nodes.

1.7 Thesis Overview

The remainder of this thesis is organized as follows.

Chapter 2 discusses the design of R-DHT and its Chord-based implementation called R-Chord. We first present *read-only mapping*, the main concept in R-DHT. Next, we discuss how the read-only mapping is applied to Chord, which results in read-only Chord (R-Chord). Subsequently, we present the optimizations to R-Chord lookup operations, i.e. routing by segments and shared finger tables. This is followed by the maintenance of R-Chord overlay network which exploits finger flexibility through backup fingers. We evaluate the performance of R-DHT through theoretical and simulation analysis.

Chapter 3 presents a hierarchical R-DHT that reduces the overhead of overlay-network maintenance. We discuss the design of hierarchical R-DHT where nodes in the top-level overlay network are organized into a Chord ring. Then, we present an approach to detect collisions in the hierarchical R-DHT by piggybacking *periodic stabilizations*, followed by two approaches to resolve the collisions, namely *supernode initiated* and *node initiated*. Lastly, we present a simulation analysis on hierarchical R-DHT.

Chapter 4 discusses Midas, a scheme to support multiple-attribute range queries on R-DHT. Midas uses Hilbert space-filling curve as the *d*-to-*one* mapping function.

We describe the indexing scheme of Midas which maps resources to R-DHT nodes, followed by Midas query engine which searches for resources that satisfy a given query. We evaluate our approach through simulations.

Finally, Chapter 5 summarizes the results of this thesis and discusses some issues that require further investigation.

Chapter 2

Read-only DHT: Design and Analysis

A distributed hash table realizes the mapping of keys onto nodes through the store operation. As a result, key-value pairs are distributed across the overlay network [14, 99, 116, 123, 133]. Data-item distribution reduces node autonomy in the aspect of key-value-pairs placement. This leads to the issues of data-item ownership and conflicting self-interest among administrative domains. In this chapter, we present R-DHT, a DHT scheme that does not distribute data items across its overlay network. We start with the terminologies and notations used throughout this chapter, followed by an overview of R-DHT. Then, we present the design of R-DHT using Chord [133] as the underlying overlay graph. This is followed by theoretical analysis and experimental evaluation, through simulations, on R-DHT lookup performance and maintenance overhead. Finally, we conclude this chapter with a summary.

2.1 Terminologies and Notations

In this section, we introduce the terms *resource type*, *host*, and *node*.

Definition 2.1. *A resource* type *is the list of attribute names of a resource. The resource type determines the* key *assigned to a resource. There could be many resource instances with the same type; these resources are assigned the same key.*

Definition 2.2. *A* host *refers to a physical entity that shares resources. Let T_h denote the set of unique keys (i.e. resource types) owned by a host whose host identifier is h.*

Figure 2.1 illustrates how the above terminologies are applied to a computational grid [47, 80]. In this example, host refers to the MDS server [4]. The two keys, $T_3 = \{2, 9\}$, denote that host 3 indexes two types of resources shared by administrative domain 3. One resource type consists of three resource instances (e.g. machines), each of which is identified by key 2. The other resource type refers to a resource whose key is 9. Details on assigning a key to a resource is discussed in Chapter 4.

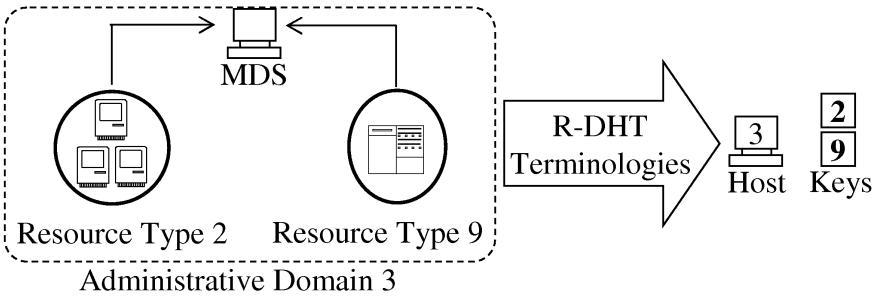


Figure 2.1: Host in the Context of Computational Grid

Definition 2.3. *A* node *refers to a logical entity in an overlay network.*

In terms of set theory, a multiple-valued function, $virtualize : hosts \rightarrow nodes$, describes the relationship between hosts and nodes (Figure 2.2). A host joins an

overlay network as one or more nodes, i.e. by assuming one or more identities in the overlay. Clearly, each node corresponds to only one host. The notion of "nodes" is equivalent to "virtual servers" in Cooperative File System [42] or "virtual hosts" in Apache HTTP Server [1].

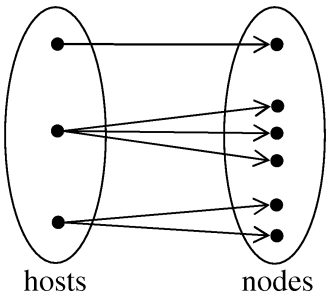


Figure 2.2: $virtualize : hosts \rightarrow nodes$

Table 2.1 shows some of the important variables maintained by each host and Chord node. In addition to its node identifier (n), each node maintains its states in an overlay topology (*finger*, *successor*, and *predecessor*).

Entity	**Variable**	**Description**
Host h	T_h	a set of unique keys owned by host h
Node n	$finger[1 \ldots F]$ *successor* *predecessor*	a finger table of F entries the next node in the ring overlay, i.e. $finger[1]$ the previous node in the ring overlay

Table 2.1: Variables Maintained by Host and Node

In presenting pseudocode, we adopt the notations from [133]:

1. Let h denote a host or its identifier, and n denotes a node or its identifier, as their meaning will be clear from the context.

2. Remote procedure calls or variables are preceded by the remote node identifier, while local procedure calls and variables omit the local node identifier.

2.2 Overview of R-DHT

R-DHT is a read-only DHT where a node supports "read-only" accesses to its keys, and does not allow keys belonging to other nodes to be written (mapped) on it. The read-only property ensures that keys are mapped onto their originating node. With its read-only property, R-DHT addresses issues such as node autonomy in placing key-value pairs, prevents stale data items, and increases lookup resiliency to node failures without the need to replicate keys.

As shown in Figure 2.3, R-DHT achieves the read-only mapping by virtualizing a *host* into a number of *nodes* where each node represents a unique key shared by the host. The node identifier space is divided into two sub-spaces: a key space and a host identifier space. This ensures the uniqueness of node identifiers without compromising R-DHT's support for a flat naming scheme [22]. Shared resources of the same type is identified by the same key and forms a segment on the overlay graph. A segment-based overlay reduces lookup path length and improves lookup resiliency to node failures without a need to replicate data items.

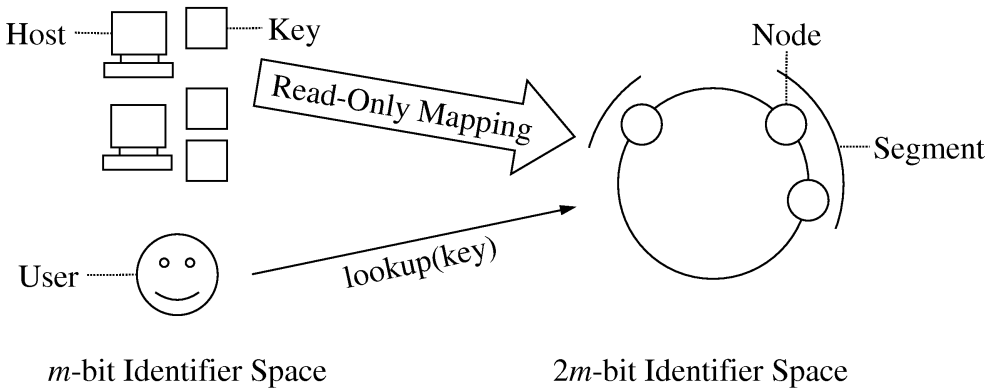


Figure 2.3: Proposed R-DHT Scheme

As an example, we discuss how R-DHT supports distributed resource discovery in a computational grid. A computational grid facilitates the sharing of compute

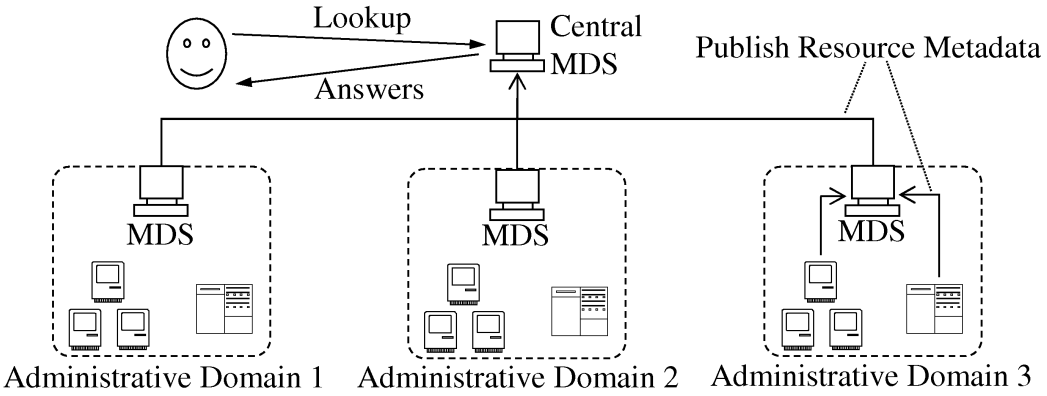
resources from different administrative domains [47, 80]. Typically, grid users search for specific resources where their job will be executed [106]. In a centralized scheme, an MDS [4] indexes all resources from all administrative domains and processes all user queries (Figure 2.4a). As the adoption of grid increases, the central MDS becomes a potential bottleneck and a single point of failure. Recently, there is a growing interest in studying the use of DHT-based resource discovery for large computational grids [27, 28, 132, 145]. Instead of depending on a third-party central MDS, DHT-based schemes distribute queries across administrative domains organized as nodes in an overlay network (Figure 2.4b). With R-DHT as the basis, a computational grid supports scalable distributed resource discovery with high result guarantee, while preserving the autonomy of administrative domains where each administrative domain stores its own resource metadata.

2.3 Design

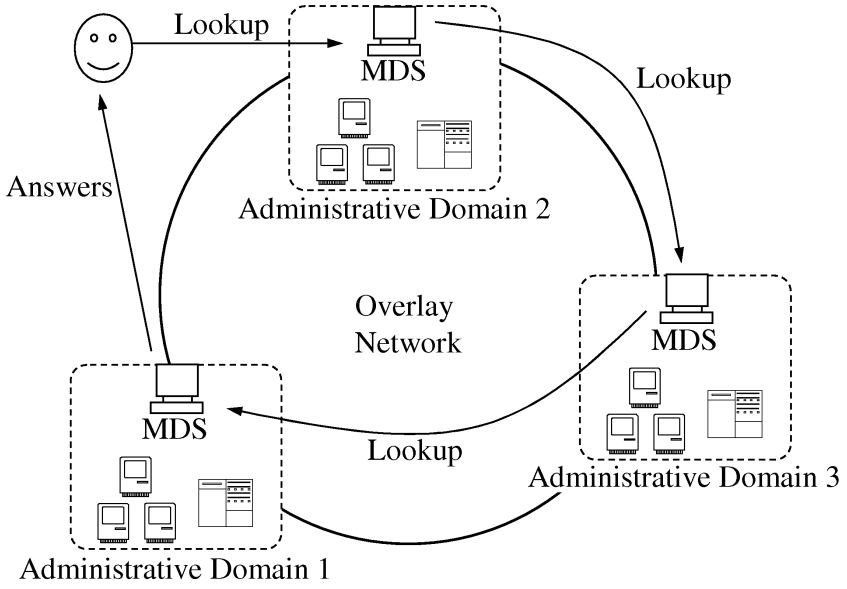
In this section, we present the design of R-DHT. We first describe read-only mapping, the main concept in R-DHT. Then, we discuss the construction of R-DHT overlay and the lookup algorithm using Chord [133] as the underlying overlay graph. An alternative of R-DHT using CAN is described in Appendix A. Subsequently, we use "R-DHT" to refer to read-only DHT in general, and "R-Chord" to refer to a Chord-based R-DHT.

2.3.1 Read-only Mapping

The basic idea of our proposal is to exploit DHT mapping whereby a key can be mapped onto a specific node if the identifier of the node is equal to the key (Figure 2.5). Thus, each node in R-DHT is a bucket with one unique key, as opposed to conventional DHT where each node is a bucket with a number of unique keys. R-DHT realizes the key-to-node mapping through virtualization



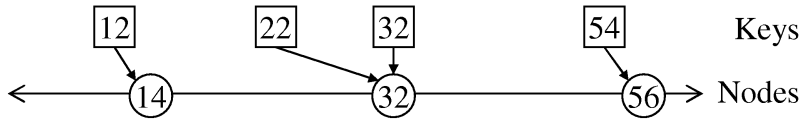
(a) Centralized



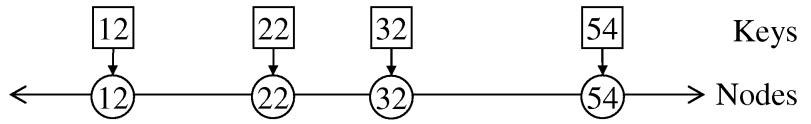
(b) Decentralized

Figure 2.4: Resource Discovery in a Computational Grid

and splitting of node identifiers. Virtualization ensures that each host can share different keys, whereas the splitting of node identifiers prevents node collisions when several hosts share the same key.



(a) Conventional DHT Maps Key to Node Identifier Closest to the Key



(b) R-DHT Maps Key to Node Identifier Equal to the Key

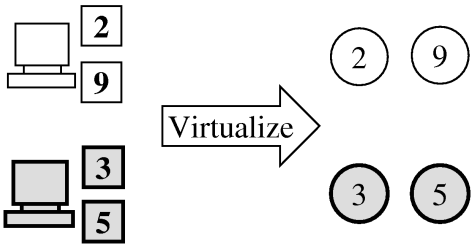
Figure 2.5: Mapping Keys to Node Identifiers

R-DHT virtualizes each host into a number of nodes by associating node n to each unique key k (i.e. a resource type) belonging to host h. Figure 2.6a shows an example of virtualizing of two hosts into four nodes, where each host with two unique keys is virtualized into two nodes. By making the node identifier equals to its associated key, R-DHT ensures that keys are not distributed. However, when nodes and keys share the same identifier space, virtualizing several hosts sharing the same key results in collisions of node identifiers (Figure 2.6b).

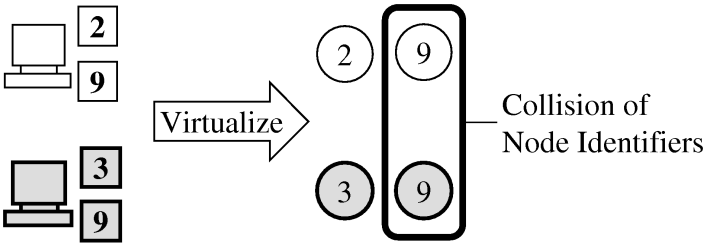
To avoid the collision of node identifiers, a node associated to key k shared by host h is assigned $k|h$ as its identifier. Each node can be uniquely identified by its node identifier which is the concatenation of the key (k) and the host identifier[1] (h). Thus, we split the node identifier space into two sub-spaces: key space and host identifier space. Figure 2.7a shows an example of node identifier where the key and the host identifier are of the same bit-length[2], i.e. ($m/2$) bits. We divide

[1] A host identifier can be derived by hashing the host's IP address or an identifier obtained from a certification authority [32].

[2] In general, the key space and the host identifier space need not be of the same size.



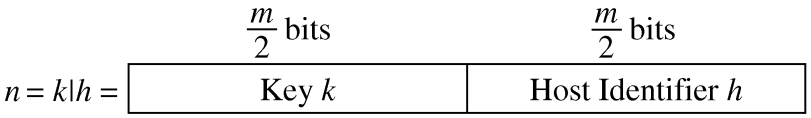
(a) Virtualize Two Hosts into Four Nodes



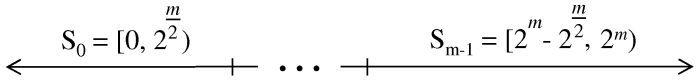
(b) Collision of Node Identifiers

Figure 2.6: Virtualization in R-DHT

the m-bit node identifier space into $2^{m/2}$ *segments*. Each segment S_k consists of $2^{m/2}$ consecutive node identifiers prefixed with k (Figure 2.7b). Therefore, each segment represents resources of the same type shared by different hosts. Segment indexing reduces lookup path length and improves fault-tolerance.



(a) m-bit Node Identifier



(b) Segmenting Node-Identifier Space

Figure 2.7: R-DHT Node Identifiers

R-DHT is designed to maintain API compatibility with conventional DHT and supports the flat naming scheme [22]. As shown in Table 2.2, R-DHT API requires the same arguments as its DHT counterparts. The *lookup*(k) API of R-DHT

supports a flat naming scheme by abstracting away the location of keys in its argument. With the flat naming scheme, queries are formulated as "find resource type k". This allows R-DHT to fully provide a hashtable abstraction where only the key are required to retrieve key-value pairs. However, systems that supports only a hierarchical naming scheme do not fully provide a hashtable abstraction of locating key-value pairs. The hierarchical naming scheme requires users to specify the location of the key as an argument to lookup operation, in addition to the key itself. Thus, queries are formulated as "find resource type k from host h", which are reminiscent of HTTP requests: "retrieve `index.html` from `www.comp.nus.edu.sg`".

Operation	**API**	
	DHT	**R-DHT**
Host h joins overlay through existing host e	$h.join(e)$	$h.virtualize(e)$
Host h shares new key k	$h.store(k)$	$h.newKey(k)$
Users at host h search for key k	$h.lookup(k)$	$h.lookup(k)$

Table 2.2: Comparison of API in R-DHT with Conventional DHT

R-DHT supports both a one-level overlay network (i.e. *flat* R-DHT) or a two-level overlay network (i.e. *hierarchical* R-DHT). In the remainder of this chapter, we discuss flat R-DHT in the following aspects: construction of overlay, lookup algorithm, and maintenance of overlay. Hierarchical R-DHT will be discussed in Chapter 3.

2.3.2 R-Chord

R-Chord is a *flat* R-DHT that organizes nodes as a (one-level) Chord overlay. Figure 2.8 presents how a new host joins R-Chord, and how an existing host shares a new key. Each new node join the ring overlay using Chord's join protocol (line 5 and 13 in Figure 2.8). Nodes are organized as a logical ring in clock-wise

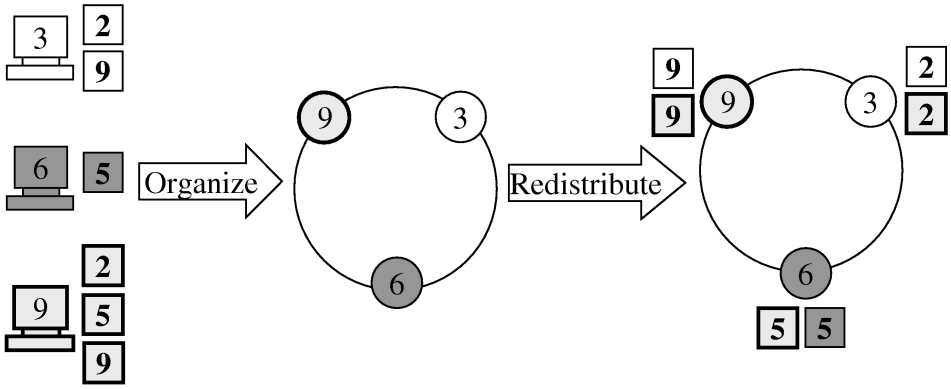
ascending order where each node identifier $(k|h)$ is interpreted as an integer. Let N denote the number of hosts, and $K = |\bigcup_{h=0}^{N-1} T_h|$ denote the total number of unique keys in the system. R-Chord divides its ring overlay into K segments where segment S_k consists of nodes prefixed with k. Thus, each segment S_k represents resources of the same type (i.e. all resources with key k) shared by different hosts.

```
1.   // h joins R-DHT through an existing host e
2.   h.virtualize(e)
3.       for each k ∈ T_h do
4.           n = k|h;
5.           n.join(e);   // Chord's protocol [133]
6.   // h shares a new key k
7.   h.newKey(k)
8.       if k ∈ T_h
9.           return;
10.
11.      T_h = T_h ∪ {k};
12.      n = k|h;
13.      n.join(h);   // Chord's protocol [133]
```

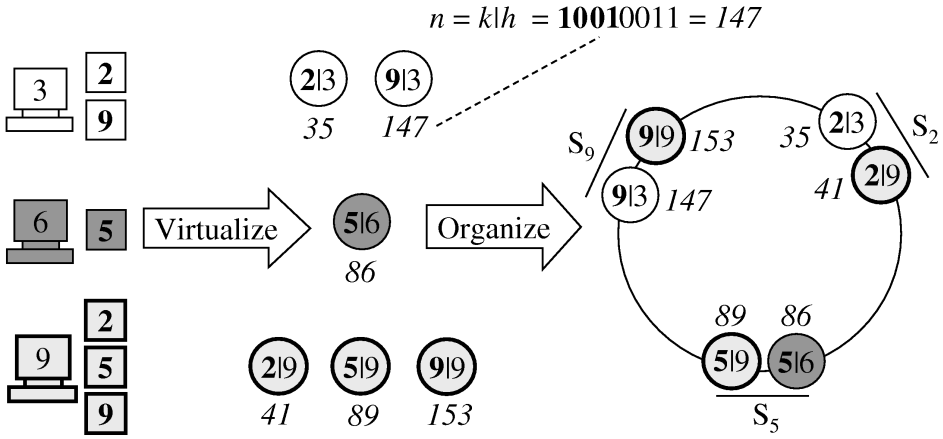
Figure 2.8: Virtualizing Host into Nodes

Figure 2.9 compares Chord and R-Chord in a grid consisting of three administrative domains (hosts), assuming that keys and host identifiers are 4-bit long. In Chord (Figure 2.9a), each host becomes a Chord node and its keys are distributed to another node whose identifier immediately succeeds the key. For example, all keys with identifier 2 will be stored on node 3. In R-Chord (Figure 2.9b), each key is mapped to its originating node. In this example, host 3 owns two unique keys, i.e. $T_3 = \{2, 9\}$, and thus, we virtualize host 3 into two nodes with node identifiers $2|3 = 35$ and $9|3 = 147$. Similarly, we virtualize host 6 and host 9 into one node and three nodes, respectively. We then organize the six nodes as an R-Chord ring based on their integral node identifier. The R-Chord ring consists of three segments, namely segment S_2 with node $2|3$ and node $2|9$, segment S_5 with node $5|9$ and $5|6$, and segment S_9 with node $9|3$ and node $9|9$. These three

segments represents three keys (resource types): key 2, key 5, and key 9.



(a) Chord Distributes Data Items among Three Nodes



(b) R-Chord Virtualizes Three Hosts into Six Nodes

Figure 2.9: Chord and R-Chord

R-DHT prevents stale data items when updated by their originating node, while conventional DHT must route the update to the node where the data item is distributed. Before the update reaches the node, the data item becomes stale. Referring to Figure 2.9, when host 3 updates its key 9, Chord routes the update to node 9 which stores the key. On the other hand, in R-Chord the update is reflected immediately because key 9 is mapped onto node 9|3 which is associated with host 3 itself.

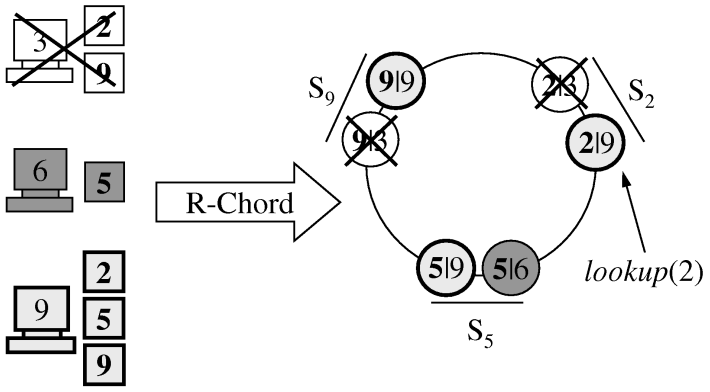
By design, R-DHT is inherently fault tolerant without a need to incorporate data-

item replications as a resiliency mechanism. Firstly, a node failure does not affect keys belonging to other nodes. Secondly, *lookup*(k) can still be successful since R-DHT can route the lookup request to other alive nodes in segment S_k. Figure 2.10 shows an example when administrative domain 3 with two shared resource types is down. Since administrative domain 9 is still alive, R-Chord can still locate resource type 2 by routing a *lookup*(*2*) request to node 2|9 (Figure 2.10a). On the contrary, the lookup fails in Chord because it is routed to node 3 (Figure 2.10b). Typically, conventional DHT replicates keys to improve its resiliency. However, this increases the risk of stale data items, i.e. data items pointing to unavailable resources in the inaccessible administrative domain 3 (Figure 2.10c).

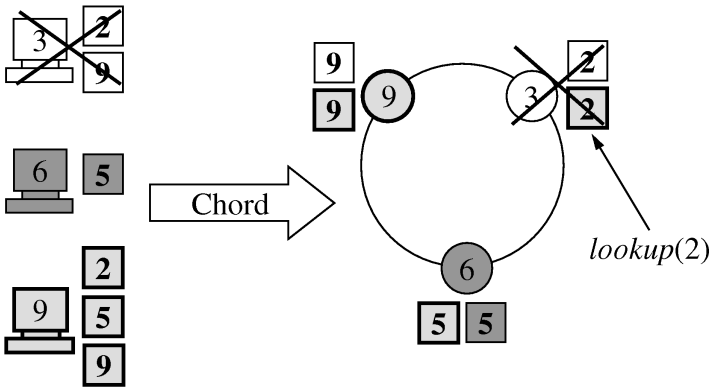
2.3.3 Lookup Optimizations

R-DHT supports a flat naming scheme where users need to specify only key k when searching. R-DHT bases its lookup on the underlying overlay's lookup protocol. With Chord as the underlying overlay, each node maintains at most m unique fingers (Figure 2.11). Lookup for key k implies locating the successor of $k|0$, i.e. the first node in segment S_k. Figure 2.12 shows a direct application of Chord lookup algorithm on R-Chord. If a lookup returns a node n' where $prefix(n') = k$, then key k is successfully found (line 11); otherwise, the key does not exist (line 14).

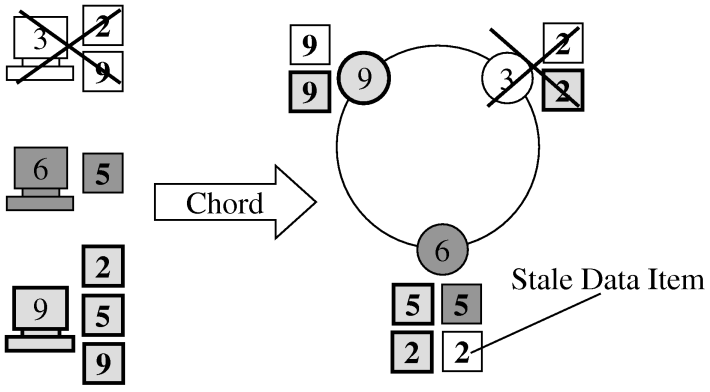
The direct application of Chord's lookup is not efficient because it does not exploit the advantages of read-only mapping. In a system with N hosts where each host has $T = \frac{\Sigma_{h=0}^{N-1}|T_h|}{N}$ unique keys on average, R-Chord consists of $N \cdot T$ nodes and hence, its lookup path length is $O(\log NT)$ hops (see Theorem 2.1). To reduce the lookup path length, R-Chord exploits the read-only mapping scheme by incorporating two optimizations, namely *routing by segments* and *shared routing tables*. The complete algorithm is shown in Figure 2.13.



(a) R-Chord

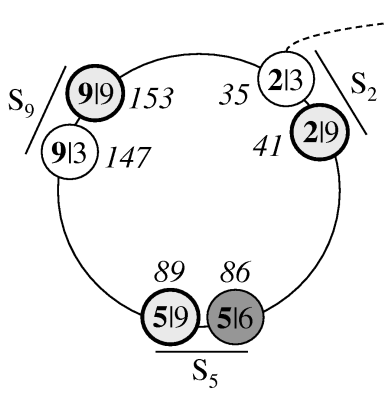


(b) Chord without Replication



(c) Replication Introduces Stale Data Items

Figure 2.10: Node Failures and Stale Data Items



Finger Table of Node **2**|3

35 + *1*	successor(36) = 41
35 + *2*	successor(37) = 41
35 + *4*	successor(39) = 41
35 + *8*	successor(43) = 86
35 + *16*	successor(51) = 86
35 + *32*	successor(67) = 86
35 + *64*	successor(99) = 147
35 + *128*	successor(163) = 41

Figure 2.11: The Fingers of Node 2|3

```
1.   // Ask h to find a node in segment S_k
2.   h.lookup(k)
3.       k' = a key randomly chosen from T_h;
4.       n = k'|h;
5.
6.       return n.node_lookup(k);
```

```
7.   // Ask n to find a node in segment S_k
8.   n.node_lookup(k)
9.       if k == prefix(n.successor) then
10.          // n's successor shares k
11.          return n.successor;
12.
13.      if n < k|0 < n.successor then
14.          return n.successor;
15.
16.      n' = n.closest_preceding_node(k|0);
17.      h' = suffix(n');
18.      return h'.lookup(k);
```

(a) Main Algorithm

```
1.   // Ask n to find the closest predecessor of id.
2.   n.closest_preceding_node(id)
3.       for i = m downto 1 do
4.           if (n < finger[i] ≤ id) then
5.               return finger[i];
6.
7.       return n;
```

(b) Helper Functions

Figure 2.12: Unoptimized R-Chord Lookup

```
1.  // Ask h to find a node in S_k
2.  h.lookup(k)
3.      for each j ∈ T_h do
4.          if j == k then
5.              return k|h;  // h is in S_k
6.
7.      n =find_segment_in_fingers(k);
8.      if n ≠ NOT_FOUND then
9.          return n;   // n is in the preceding segment of S_k
10.
11.     for each j ∈ T_h do
12.         n = j|h;
13.         if n < k|0 < n.successor then
14.             return n.successor;
15.
16.     n = closest_preceding_node(k);
17.     h' = suffix(n);
18.     return h'.lookup(k);
```

(a) Main Algorithm

```
1.  // Ask h to find a finger pointing to S_k
2.  h.find_segment_in_fingers(k)
3.      for each j ∈ T_h do   // Iterate all local nodes
4.          n = j|h;
5.          for i = 1 to m do
6.              if prefix(n.finger[i]) == k then
7.                  return n.finger[i];
8.
9.      return NOT_FOUND
```

```
10. // Ask h to find the closest predecessor of id.
11. h.closest_preceding_node(id)
12.     x = id + 1;  // Initialize to the farthest predecessor
13.
14.     for each k ∈ T_h do
15.         n = k|h;
16.         for i = m downto 1 do
17.             f = n.finger[i];
18.             if (n < f < id)    // Is f the closest predecessor known by node n?
19.                 and (x < f < id) then    // Is f a closer predecessor than x?
20.                 x = f;
21.
22.     return x;
```

(b) Helper Functions

Figure 2.13: R-Chord Lookup Exploiting R-DHT Mapping

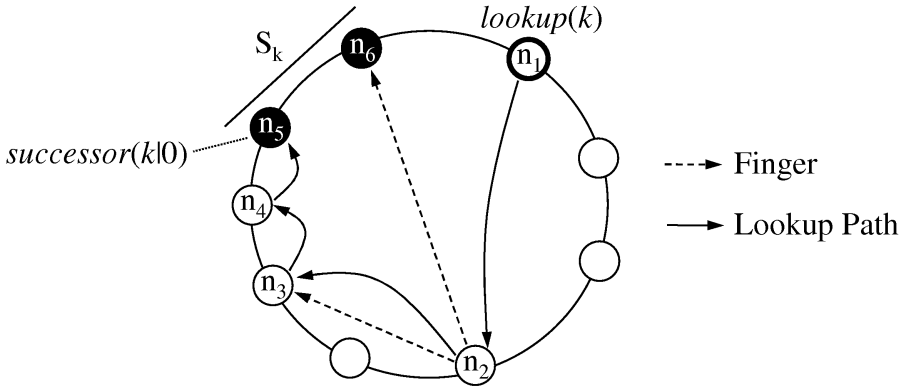
2.3.3.1 Routing by Segments

R-DHT divides its node-identifier space into segments, with each segment representing a key. Therefore, locating key k equals to locating any node within segment S_k, instead of strictly locating $successor(k|0)$ only. R-Chord exploits this segmentation by forwarding a $lookup(k)$ request from one segment to another segment (line 4, 7, and 16 in Figure 2.13a, and line 6 in Figure 2.13b). Each routing step halves the distance, in term of *segments*, to the destination segment S_k (see Lemma 2.2 for the proof). As such, in a system with K segments, routing by segments reduces lookup path length to $O(\log K)$. Since segments are identified by the prefix of node identifiers, our routing-by-segments scheme is essentially a prefix-based routing optimization applied for the Chord lookup protocol.

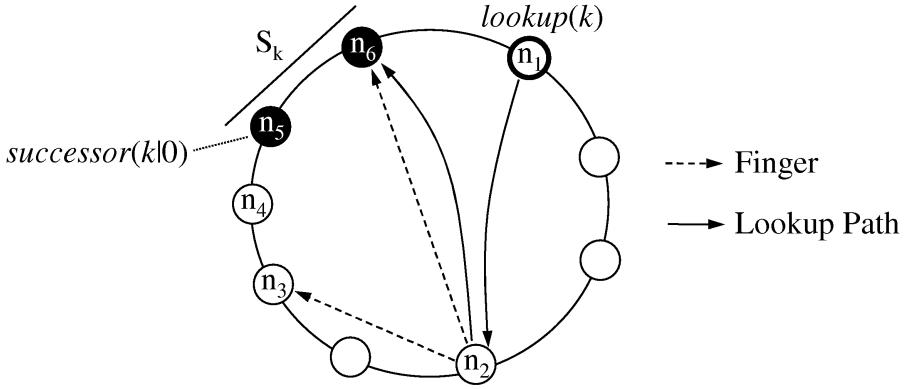
Figure 2.14 illustrates the processing of a $lookup(k)$ request initiated by node n_1. Without routing by segments, i.e. the direct application of Chord's lookup protocol, the lookup path is $n_1 \rightarrow n_2 \rightarrow n_3 \rightarrow n_4 \rightarrow n_5$ because we always locate node $n_5 = successor(k|0)$. However, with routing by segments, we realize that one of the intermediate hops, node n_2, has a finger pointing to n_6. Though node n_6 is not $successor(k|0)$, it also shares key k and is in segment S_k. Since the lookup can then be completed at node n_6, the optimized lookup path becomes $n_1 \rightarrow n_2 \rightarrow n_6$.

2.3.3.2 Shared Finger Tables

To limit the lookup path length at $O(\log N)$ hops even when $K > N$, our routing algorithm utilizes all the $|T_h|$ finger tables maintained by each host h (line 3 and 11 in Figure 2.13a, and line 3 and 14 in Figure 2.13b). In other words, a node's finger table is shared by all nodes from the same host. As such, visiting one host is equivalent to visiting all the $|T_d|$ nodes which correspond to the host (Figure 2.15). The proof that this optimization leads to $O(\log N)$-hop lookup path length, similar to Chord, is presented in Theorem 2.2. However, the intuitive explanation is as



(a) No Routing by Segments



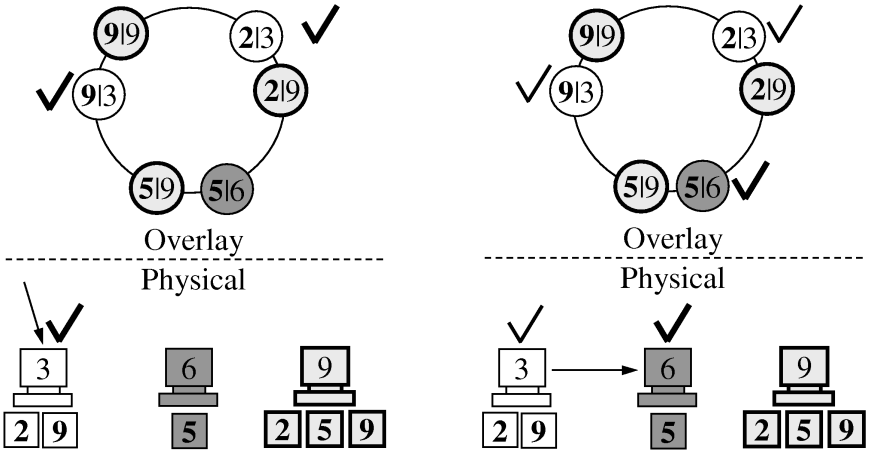
(b) Routing by Segments

Figure 2.14: *lookup*(k) with and without Routing by Segments

follows: since the distance between any two points in the overlay is at most N *hosts*, it takes $O(\log N)$ hops to locate any segment. Thus, even though the number of nodes in the overlay is greater than N due to host virtualization, the lookup path length is not affected.

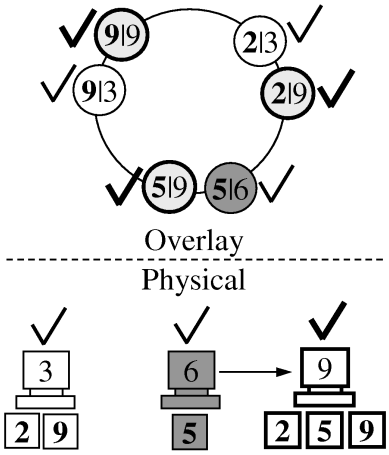
2.3.4 Maintenance of Overlay Graph

As with Chord, R-Chord maintains its overlay through periodic stabilizations. The periodic stabilization is implemented as two functions: *stabilize_successor*() and *correct_fingers*(). The first function corrects successor pointers, i.e. the first finger, in addition to predecessor pointers, whereas the later correct the remaining fingers in a finger table. The rate in which these functions are invoked is an



(a) Visiting Host 3 is Equal to Visiting Node 2|3 and Node 9|3 in the Overlay

(b) Visiting Host 6



(c) Visiting Host 9 — All Nodes are Visited

Figure 2.15: Effect of Shared Finger Tables on Routing

implementation matter.

During the periodic stabilization, R-Chord exploits *finger flexibility* which is inherent in our segment-based overlay. Finger flexibility denotes the amount of freedom available when choosing a finger. A higher finger flexibility increases the robustness of lookup since finger tables deplete slower in the event of node failures [62]. Finger flexibility also allows proximity-based routing to reduces lookup latency.

However, our current R-Chord implementation has yet to exploit this feature.

The segment-based overlay improves the finger flexibility of R-Chord whereby $n.finger[i]$ is allowed to point to any nodes in the same segment as $successor(n + 2^{i-1})$ – this is an improvement over $O(1)$ finger flexibility of Chord[3]. To exploit this finger flexibility, we employ the *backup fingers* scheme which is reminiscent of Kademlia's κ-bucket routing tables [99]. With such a scheme, every R-Chord node n maintains *backups* for each of its fingers. Thus, when f dies (i.e. points to a dead node), n still has a pointer to segment $S_{prefix(f)}$. The new structure of R-Chord finger table is shown in Figure 2.16. We use $n.finger[i]$ and $backup(n.finger[i])$ to denote the main finger and the list of backup fingers, respetively.

Description	**Finger**	**Backups**
1^{st} finger	$a\|x$	$a\|y$, $a\|z$
2^{nd} finger	$b\|t$	$b\|r$, $b\|s$, $b\|u$
...	...	...

Figure 2.16: Finger Tables with Backup Fingers

Figure 2.3.4 shows the algorithm to correct successors in R-Chord. When a new successor is detected, the old successor pointer is added into the backup list instead of being discarded (line 7). Similarly, when a new predecessor is to be set, the old predecessor is also added into the backup list (line 19). We then ensure that the backup list of successors and predecessors contains only nodes with the same prefix as the new successor (line 9) and predecessor (line 21), respectively.

Figure 2.18 shows that the algorithm to correct finger f in R-Chord exploits finger flexibility. When a new finger is added, we also construct its backup list based on the entries piggybacked from the remote node and the older valid backup

[3]To improve its robustness in spite of its lower finger flexibility, each Chord node caches additional entries in its finger tables. Such scheme can also be adopted in R-Chord.

```
1.  // n verifies its successor pointer, and announces itself to the successor.
2.  n.stabilize_successor()
3.      p = successor.predecessor;
4.      if n < p < successor then
5.          // Prepare a new backup list
6.          backup(successor) = backup(successor) ∪ successor
7.                                  ∪ backup(p);
8.          Remove backup(successor) entries whose prefix ≠ prefix(p);
9.
10.         successor = p;    // New successor pointer
11.         successor.notify(n);
12. // n' thinks it might be our predecessor
13. n.notify(n')
14.     if (predecessor == nil) or (predecessor < n' < n) then
15.         // Prepare a new backup list
16.         backup(predecessor) = backup(predecessor) ∪ predecessor
17.                                 ∪ backup(n');
18.         Remove backup(predecessor) entries whose prefix ≠ prefix(n');
19.
20.        predecessor = n';    // New predecessor pointer
```

Figure 2.17: Successor-Stabilization Algorithm

entries (line 7–9 in Figure 2.18a). As with lookups, the finger-correction algorithm incorporates shared finger tables (line 4 in Figure 2.18a and line 3 in Figure 2.18b).

2.4 Theoretical Analysis

In this section, we analyze the lookup performance of R-Chord, and compare the overhead of the mapping scheme in R-Chord and conventional Chord-based DHT (hereafter referred simply as Chord). Let N denote the number hosts, T_h denote the set of unique keys owned by host h, T denote the average number of unique keys owned by each host (i.e. $\frac{\Sigma_{h=0}^{N-1}|T_d|}{N}$), and K denote the total number of unique keys in the system (i.e. $|\bigcup_{h=0}^{N-1} T_h|$). The Chord overlay consists of N nodes (one node per host) and the R-Chord overlay consists of V ($= NT$) nodes (on average, T nodes per host).

```
1.  // Node n to correct its i^th finger.
2.  n.correct_finger(i)
3.      f = n + 2^(i-1);
4.      h = suffix(n);
5.      f' = h.find_successor(f);
6.
7.      // Prepare a new backup list
8.      backup(finger[i]) = backup(finger[i]) ∪ finger[i] ∪ backup(f');
9.      Remove backup(finger[i]) entries whose prefix ≠ prefix(f');
10.
11.     finger[i] = f';    // New finger
```

(a) Main Algorithm

```
1.  // Ask host h to find successor(id).
2.  h.find_successor(id)
3.      for each k ∈ T_h do
4.          n = k|h;
5.          if n < id ≤ n.successor then
6.              return n.successor;
7.
8.      n = closest_preceding_node(id);   // See Figure 2.13b
9.      return n'.find_successor(id);
```

(b) Helper Functions

Figure 2.18: Finger-Correction Algorithm

2.4.1 Lookup

In order to analyze the lookup performance in R-Chord, we first present the proof on Chord lookup path length (see also Theorem 2 in [133]).

Theorem 2.1. *In an N-node Chord overlay, the lookup path length is $O(\log N)$.*

Proof. Suppose that node n wishes to locate the successor of k. Let p be the node that immediately precedes k. To prove the lookup path length, we first show that each routing step halves the distance to p.

If $n \neq p$, then n forwards a lookup request to the closest predecessor of k in its

finger table. If p is located in the interval $[n + 2^{i-1}, n + 2^i)$, then n will contact its i^{th} finger, the first node f in the interval. The distance between n and f is at least 2^{i-1}, and the distance from f to p is at most 2^{i-1}. This also means that the distance from f to p is at most half of the distance from n to p.

Assume that node identifiers are uniformly distributed in the 2^m circular identifier space, the number of forwarding necessary to locate k will be $O(\log N)$, which is explained as follows. After i forwarding, there are $N/2^i$ remaining nodes to choose for the next forwarding. Thus, with $\log N$ forwarding, there is only one node as the next hop; this node is the successor of k. □

In the following, we present the proof on the lookup path length of R-Chord utilizing two intermediate results, Lemma 2.1 and Lemma 2.2.

Lemma 2.1. *The probability that host h owns key k, i.e. $P(k \in T_h)$, is bounded by* $\ln \frac{K}{K-T}$.

Proof. Let $k \in T_h$ denote key k owned by host h. We define the probability that host h owns key k as

$$\begin{aligned} P(k \in T_h) &= P(e_1) + P(e_2|\overline{e_1}) + P(e_3|\overline{e_1}, \overline{e_2}) + \ldots + P(e_T|\overline{e_1}, \ldots, \overline{e_{T-1}}) \\ &= \sum_{i=1}^{T} P(e_i|\overline{e_1} \ldots \overline{e_{i-1}}) \end{aligned}$$

where e_i denotes the outcome for $k_i = k$, and $\overline{e_i}$ denotes the outcome for $k_i \neq k$.

Assuming that $T \ll K$ and k is uniformly drawn from $\{1, ..., K\}$, we approximate $P(k \in T_h)$ using the first-order Markov process as follows. Firstly, we consider K resource types as K balls, each of which with a unique color. If we pick T balls sequentially, then the probability that the i^{th} outcome, where $i \leq T$, produces the

k-colored ball is $\frac{1}{K-i+1}$. This leads to the following equation:

$$P(k \in T_h) = \begin{cases} 0 & \text{if } T = 0 \\ \frac{1}{K} & \text{if } T = 1 \\ \sum_{i=1}^{T} \frac{1}{K-i+1} = \sum_{i=K-T+1}^{K} \frac{1}{i} & \text{if } T > 1 \end{cases}$$

Since $H_x = \sum_{i=1}^{x} \frac{1}{i} = \ln x + O(1)$, then

$$P(k \in T_h) = \begin{cases} 0 & \text{if } T = 0 \\ \frac{1}{K} & \text{if } T = 1 \\ H_K - H_{K-T} & \text{if } T > 1 \end{cases}$$

$$= \begin{cases} 0 & \text{if } T = 0 \\ \frac{1}{K} & \text{if } T = 1 \\ ln\ \frac{K}{K-T} & \text{if } T > 1 \end{cases}$$

□

Lemma 2.2. *Routing by segments leads to $O(\log K)$-hops lookup.*

Proof. To analyze the lookup path length due to our routing-by-segments optimization, we compare the finger tables in R-Chord and Chord.

According to Theorem 2.1, in a Chord system consisting of N nodes, the lookup path length is $O(\log N)$ if Chord is able to route a lookup request from one node to another such that each step halves the distance to the destination node. To achieve this, each node n maintains $O(\log N)$ unique fingers where:

1. The distance between n and $n.finger[i+1]$ is twice the distance between n and $n.finger[i]$.

2. The largest finger of n points to $successor(N/2)$.

We now show the similarity of finger tables in R-Chord and Chord. Let $S = O(NP(k \in T_h))$ denote the average number of nodes in a segment. In an R-Chord system consisting of V nodes, each node n maintains $O(\log V)$ unique fingers where:

1. The first $O(\log S)$ of the $O(\log V)$ fingers point to the segment containing $n.successor$. The remaining $O(\log V - \log S)$ fingers point to $O(\log V - \log S)$ different segments because the distance between n and $n.finger[\log(S+j+1)]$ is twice the distance between n and $n.finger[\log(S+j)]$, where $0 \leq j \leq \log N - \log S$.

2. The largest finger of n will point to $successor(N/2)$, which is a node in the segment that succeeds segment $K/2$.

Using the same argument as in Chord, R-Chord routes a lookup request from one segment to another and each hop halves the distance, *in terms of the number of segments*, to the destination segment. Since R-Chord consists of K segments, a lookup will cost $O(\log K)$ hops. □

Theorem 2.2. *With shared finger tables, the lookup path length in R-Chord is* $O(\min(\log K, \log N))$ *hops.*

Proof. If $K \leq N$ then $\log K \leq \log N$. Thus, according to Lemma 2.2, this theorem is true.

Consider $K > N$. When host h processes $lookup(k)$, we choose two consecutive keys $s, u \in T_h$ where

1. $s < k < u$

2. There is no $v \in T_h$ such that $s < v < u$

The two keys, s and u, are associated with two nodes, namely node $s|h$ and node $u|h$, respectively. The destination segment S_k will be located between node $s|h$ and node $u|h$, and the distance between $s|h$ and $u|h$ is $O(K/T)$. Since $K \leq V$, then $K/T = O(V/T) = O(NT/T) = O(N)$. Thus, according to Theorem 2.1, *lookup*(k) can be routed from node $s|h$ to segment S_k in $O(\log N)$ hops. □

Theorem 2.2 shows that the lookup performance in R-Chord is at worst comparable to the lookup performance in Chord. Due to the shared finger tables, R-Chord's lookup path length is equivalent to Chord where the number of hops to reach a certain node is affected by the number of hosts in the physical network (N) instead of the number of nodes in the overlay network (V).

2.4.2 Overhead

The following theorems compare the maintenance overhead in R-Chord and Chord in terms of the cost of virtualization, number of fingers per host, cost of updating data items, and overhead of replication.

Theorem 2.3. *The cost for a host to join R-Chord and Chord is* $O(|T_h| \log^2 V)$ *and* $O(\log^2 N + |T_h| \log N + K \ln \frac{K}{K-T})$*, respectively.*

Proof. R-Chord virtualizes a host into $|T_h|$ nodes in an overlay graph of size V. Since a node join costs $O(\log^2 V)$, the host join costs $O(|T_h| \log^2 V)$.

In Chord, a host join consists of a node-join operation, $|T_h|$ store operations to store the key-value pairs belonging the new host, and migrations of key-value pairs. The node-join operation costs $O(\log^2 N)$ and each store operations costs $O(\log N)$. The migration process moves $O(N/K)$ *unique keys* from an existing node, which is the successor of the new node, to the new node n. As there are $O(NP(k \in T_h))$ key-value pairs per unique key, the migration costs $O(K \ln \frac{K}{K-T})$.

Therefore, the host join costs $O(\log^2 N + |T_h| \log N + K \ln \frac{K}{K-T})$ in total. □

Theorem 2.3 shows that the cost to join R-Chord is higher than Chord. This is because R-Chord replaces a store operation with a join operation, and the cost of a join operation is higher than a store operation.

Theorem 2.4. *A host maintains $O(|T_h| \log V)$ and $O(\log N)$ unique fingers in R-Chord and Chord, respectively.*

Proof. In a R-Chord ring consisting of V nodes, each node maintains finger table consisting of $O(\log V)$ unique fingers. Because R-Chord virtualizes a host into $|T_h|$ nodes, the host maintains $O(|T_h| \log V)$ fingers in total.

In Chord, each host joins a ring overlay as a node. Thus, with N hosts, the Chord ring consists of N nodes where each node maintains $O(\log N)$ unique fingers. □

The following theorem shows the overhead of maintaining an overlay, i.e. stabilization cost, in terms of the number of messages sent.

Theorem 2.5. *In R-Chord, the stabilization cost to correct all fingers (including successor pointers) is $O(V \log N \log V)$ messages. Since $N \leq V$, the stabilization cost is also $\Omega(V \log^2 V)$ messages. On the other hand, the stabilization cost in Chord is $O(N \log^2 N)$ messages.*

Proof. R-Chord overlay consists of V nodes and each node maintains $O(\log V)$ fingers. Correcting the i^{th} finger of node n is performed by locating the node that succeeds $n + 2^{i-1}$; this costs $O(\log N)$ hops, i.e. shared finger tables as the only lookup optimization (see Theorem 2.2). Thus, the cost of stabilization is $O(V \log V \log N)$ hops.

In Chord, the overlay consists of N nodes and each node maintains $O(\log N)$ fingers. Correcting each finger is performed by locating $successor(n+2^{i-1})$, which costs $O(\log N)$ hops. Thus, the cost of stabilization is $O(N \log^2 N)$ hops. □

Theorem 2.4–2.5 states that a higher number of fingers implies a higher overhead in maintaining an overlay graph. This affects the scalability of R-Chord particularly when a host is virtualized into many nodes. To reduce the overhead of periodic stabilizations, which are employed by the current implementation of R-Chord to correct fingers, nodes need not to correct all their fingers each time the stabilization procedure is invoked. Instead, each invocation corrects only a subset of a node's fingers, e.g. the successor pointer and another randomly-chosen finger; this is similar to Chord's current implementation of periodic stabilizations. The drawback of this approach is the increase of the number of incorrect entries in a finger table; this increases the lookup path length. However, as long as the successor pointer, i.e. the first finger, is maintained, the lookup will still terminate at the correct node.

Theorem 2.6. *The finger flexibility in R-Chord and Chord is* $O(N \ln \frac{K}{K-T})$ *and* $O(1)$*, respectively.*

Proof. Assume that $successor(n + 2^{i-1})$ is in segment S_k, in R-Chord, the i^{th} finger of node n can point to any node in segment S_k. The number of nodes in this segment is equal to to the number of hosts that own key k, which is $O(NP(k \in T_h))$ hosts. Hence, the finger flexibility is $O(N \ln \frac{K}{K-T})$.

In Chord, the i^{th} finger of n must point to $successor(n + 2^{i-1})$, and hence, $O(1)$ finger flexibility. □

As mentioned in Section 2.3.4, a higher finger flexibility increases the robustness of lookup in the presence of node failures. Higher finger flexibility also allows

proximity-based routing to reduces lookup latency.

Theorem 2.7. *In Chord, adding a key-value pair costs* $O(\log N)$. *In R-Chord, adding a key-value pair whose key* k *already exists in host* h *(i.e.* $k \in T_h$ *before the addition) costs* $O(1)$*, while adding a key-value pair whose key is new for host* h *(i.e.* $k \notin T_h$ *before the addition) costs* $O(\log^2 V)$.

Proof. In Chord, a key-value pair is stored on the successor of the key. This costs $O(\log N)$.

In R-Chord, if $k \in T_h$ before the addition, then no new node is created and hence, the cost is $O(1)$. However, if $k \notin T_h$ before the addition, then a new node is created and joins the R-Chord system. This costs $O(\log^2 V)$. □

In applications such as P2P file sharing, sharing a new file is equal to adding a new resource type. However, in computational grid, a resource type consists of many resource instances, and an administrative domain can add new instances to one of its existing resource type. Theorem 2.7 shows that using R-Chord, the administrative domain does not need to notify other nodes in the R-Chord overlay.

Theorem 2.8. *In R-Chord, the total number of key-value pairs with the same key is* $O(N \ln \frac{K}{K-T})$. *In Chord, assuming that each key-value pair is replicated* $O(\log N)$ *times, then the total number of key-value pairs with the same key is* $O(N \ln \frac{K}{K-T} \log N)$.

Proof. Given N hosts, the number of key-value pairs with the same key is $O(NP(k \in T_h))$. Since R-Chord does not redistribute and replicate key-value pairs, the number of key-value pairs with the same key is also $O(N \ln \frac{K}{K-T})$. In Chord, because each key-value pair is replicated $O(\log N)$ times, then the total number of key-value pairs with the same key will be $O(N \ln \frac{K}{K-T} \log N)$. □

R-Chord does not need to replicate data items to improve the lookup resiliency to node failures. This eliminates the network bandwidth required to replicate data items and the complexity in maintaining consistency among replicas.

Even when data items are not replicated, data-item distribution can still lead to the problem of inconsistent data items. In conventional DHT, updates must be propagated from to the node responsible to store a data item. This is shown in the following corollary.

Corollary 2.1. *In Chord, the cost of a host to update its key-value is* $O(\log N)$. *In R-Chord, the cost is* $O(1)$.

Proof. In Chord, the cost for a host (the originating node) to propagate an update on its key-value pair to another node costs $O(\log N)$, according to Theorem 2.1.

In R-Chord, the key-value pair is mapped to its originating node. Hence, the cost of updating the key-value pair is $O(1)$. □

Corollary 2.1 shows that R-Chord improves the performance of a host in updating its data items, including the deletion of data items. In the case of computational grid, updates occur when an administrative domain changes the configuration of its shared resources, or changes the number of resource instances of a resource type.

2.4.3 Cost Comparison

Table 2.3 summarizes the performance analysis of R-Chord. We show that the lookup path length in R-Chord is shorter than Chord and in the worst case, it is equal to Chord. However, our mapping scheme increases the cost for a host to join an overlay. In addition, each host in R-Chord has more fingers to correct

due to the host being virtualized into nodes. Thus, the scalability of R-Chord is determined by the number of nodes associated to each host. When each host is virtualized into one node only, the scalability of R-Chord is equal to traditional Chord.

Property	**Chord**	**R-Chord**
Lookup a key	$O(\log N)$	$O(\min(\log K, \log N))$
Host join	$O(\log^2 N + \|T_h\| \log N + K \ln \frac{K}{K-T})$	$O(\|T_h\| \log^2 V)$
# unique fingers per host Finger flexibility	$O(\log N)$ $O(1)$	$O(\|T_h\| \log V)$ $O(N \ln \frac{K}{K-T})$
Stabilization	$O(N \log^2 N)$	$O(V \log V \log N)$
Add a key that exists Add a new key Update a key-value pair # key-value pairs with the same key	$O(\log N)$ $O(\log N)$ $O(\log N)$ $O(N \ln \frac{K}{K-T} \log N)$	$O(1)$ $O(\log^2 V)$ $O(1)$ $O(N \ln \frac{K}{K-T})$

Table 2.3: Comparison of Chord and R-Chord

2.5 Simulation Analysis

In this section, we evaluate R-DHT by simulating an R-Chord-based resource indexing and discovery scheme in a large computational grid. As illustrated in Figure 2.4b, a computational grid consists of many administrative domains, each of which share one or more compute resources. Each administrative domain, represented by its MDS server (i.e. host), joins an R-Chord overlay and stores only its own resource metadata (i.e. data items). The key of each data item is determined by the type (i.e. attributes) of compute resource associated with the data item. Thus, the number of unique keys owned by a host denotes the number of unique resource types shared by an administrative domain.

To facilitate our experiments, we implement R-Chord using the Chord simulator

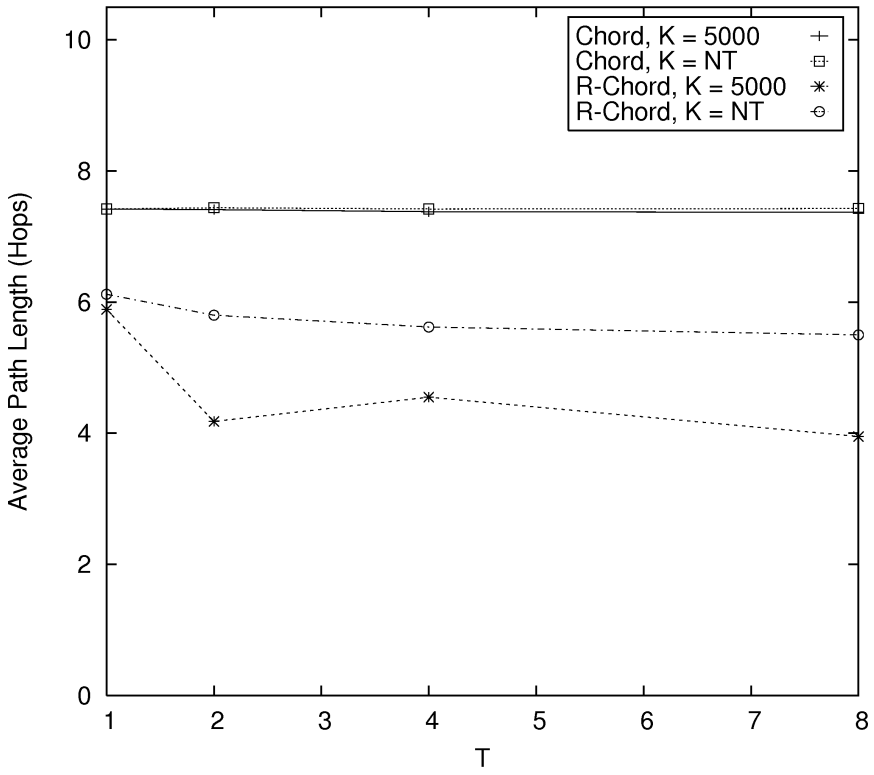
[2]. Let $m = 18$-bit unless stated otherwise. The network latency between hosts is exponentially distributed with a mean of 50 ms, and the time for a node to process a request is uniformly distributed in [5, 15] ms. In the following subsections, we compare the lookup path length, resiliency to node failures, time to correct overlay, and lookup performance on incorrect overlays.

2.5.1 Lookup Path Length

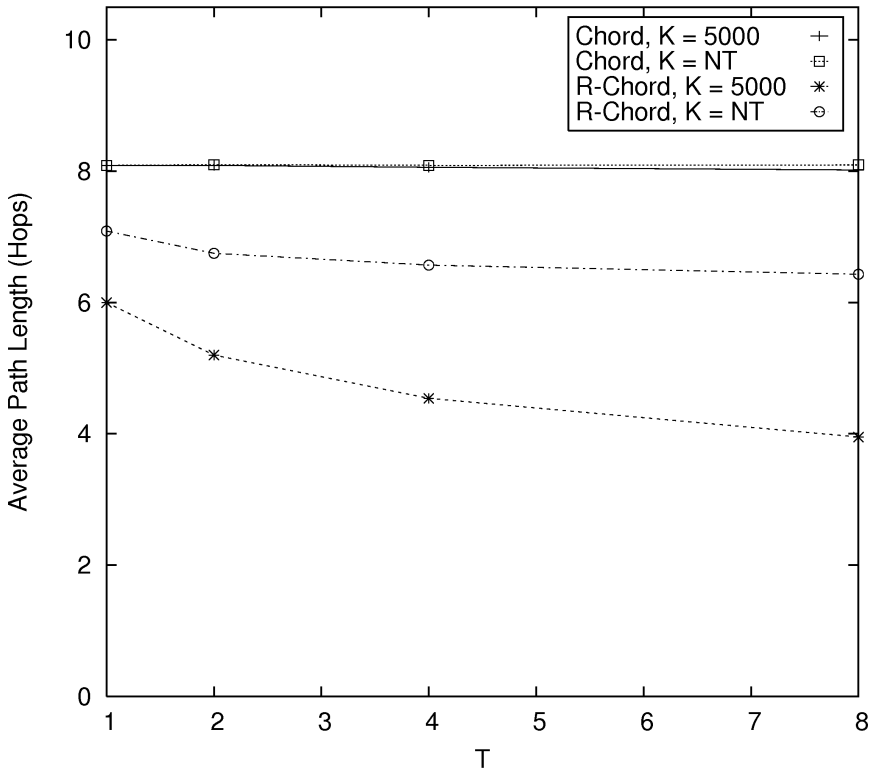
To verify Theorem 2.2, we measure the *average lookup path length* of 500,000 lookups that arrived based on a Poisson distribution with mean arrival rate $\lambda = 1$ lookup/second. Each lookup requests for a randomly selected key and is initiated by a randomly chosen host. Assuming that T denotes the average number of unique keys per host, each host has $|T_h| \sim U[0.5T, 1.5T]$ unique keys.

As shown in Figure 2.19, the average lookup path length in R-Chord is 20-30% lower than in Chord. When K $(= NT) > N$, R-Chord's overlay consists of K segments and each segment consists of one node. According to Theorem 2.2, the lookup path length of R-Chord is affected only by N, and hence, increasing K does not increase the lookup path length. However, for $K \leq N$, the lookup path length increases with K.

Figure 2.19 also shows that in R-Chord, increasing T reduces the average path length, which can be explained as follows. First, as each host maintains $O(|T_h| \log NT)$ unique fingers (Theorem 2.4), an increase in T also increases the number of fingers per hosts. Several studies such as [64, 122] also reveal that maintaining more fingers reduces the lookup path length. Secondly, an increased in T increases the number of segments occupied by a host, and hence, each host has a higher probability to be visited.



(a) $N = 10,000$ Hosts



(b) $N = 25,000$ Hosts

Figure 2.19: Average Lookup Path Length

Our simulation result confirms Theorem 2.2, i.e. when $K > N$, the lookup path length in R-Chord has the same upper bound as Chord. When $K \leq N$, the lookup path length in R-Chord is shorter than Chord.

2.5.2 Resiliency to Simultaneous Failures

To evaluate the resiliency when no churn occurs, we measure the *average lookup path length* and *failed lookups* with the following procedures. First, we setup a system of $N = 25,000$ hosts where all hosts have T unique keys on average. Then, we fail a fraction of hosts[4] simultaneously, disable the periodic finger correction after the simultaneous failures, and simulate 500,000 lookup requests with mean arrival rate $\lambda = 1$ lookup/second (Poisson distribution). We define a lookup for a key as fail if it results in (i) a *false negative* answer where existing resources (i.e. at least one originating node of the key is alive) cannot be located, or (ii) a *false positive* answer where stale data items are returned. We also assume that Chord stores a key-value pair only to $successor(key)$ and does not further replicate the key-value pair to several other nodes. Finally, we exploit the property of finger flexibility in R-Chord by maintaining a maximum of four backups per finger.

Figure 2.20 shows the average lookup path length with 25% and 50% of simultaneous host failures. For $K\ (= TN) > N$, the average lookup path length in R-Chord shows a trend similar to that of in Chord, i.e. lookup path length increases as more hosts fail (Figure 2.20a). Because each segment consists of only one node, R-Chord cannot exploit finger flexibility. Hence, as the percentage of host failures increases, the number of valid fingers, i.e. pointing to alive nodes, reduces in each node's finger table. For $K \leq N$ (Figure 2.20b), R-Chord has a shorter average path length than Chord and the lookup path length is not significantly affected by the number of failed hosts. The reason is as follow. Firstly, R-Chord provides $O(\log K)$-hops

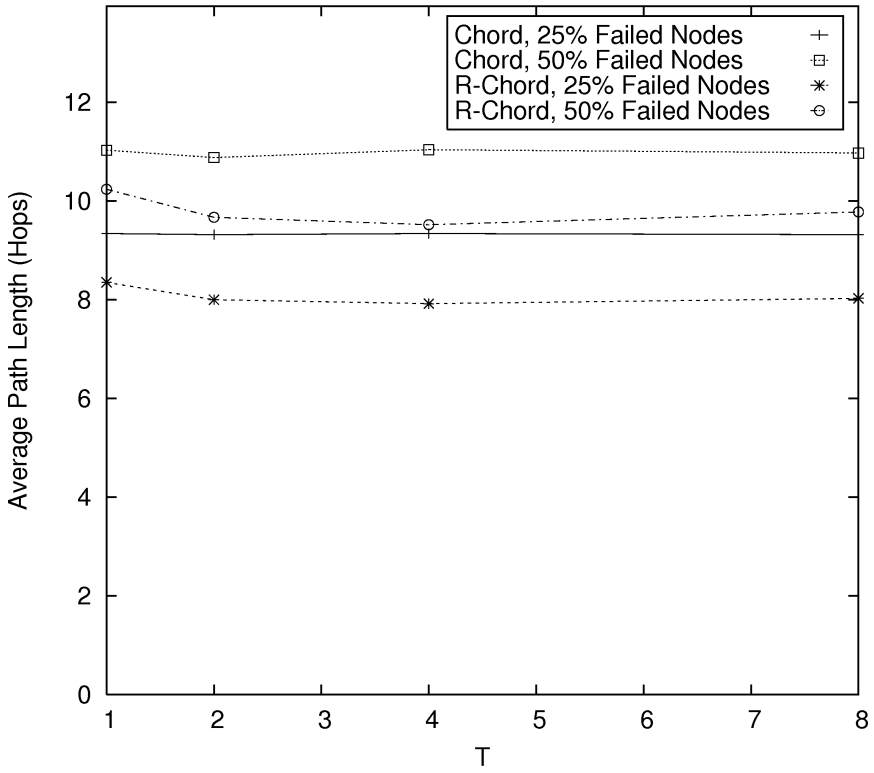
[4]In R-Chord, one host fail results in simultaneous node fails.

lookup path length only if each node has a correct finger tables. In the case of node failures, the finger table has a reduced number of valid fingers; this increases the lookup path length. However, since $K \leq N$ implies each segment consists of more than one node, R-Chord effectively exploits finger flexibility to maintain the number of valid fingers.

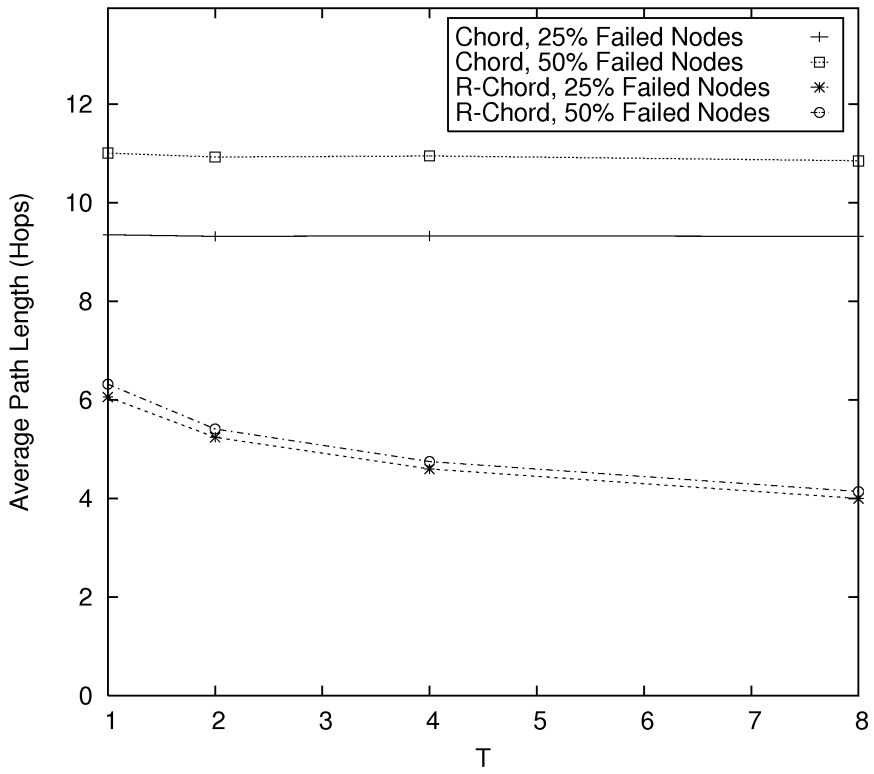
In terms of failed lookups, Figure 2.21 shows that for $K > N$ and $K \leq N$, R-Chord is 70% and 95% lower than in Chord, respectively. For $(K = NT) > N$ (Figure 2.21a), Chord has more failed lookups because key-value pairs are stored on another host. In the event of the host that stores the key-value pair fails, a lookup request to that host will be unsuccessful though the host that owns the key-value pair is still alive. For $K \leq N$ (Figure 2.21b), R-Chord achieves even less failed lookups (95% lower than Chord) because R-Chord exploits the property that each key is available in a segment consisting of several nodes. Hence, even if some of these nodes fail, R-Chord can still reach the remaining hosts in the segment through the backup fingers. Thus, R-Chord offers better resiliency to simultaneous failures in comparison to the conventional DHT.

2.5.3 Time to Correct Overlay

The correctness of an overlay network is crucial to the lookup performance in DHT. To ensure the correctness of an overlay in the event of membership changes, each node periodically corrects its fingers, i.e. periodic stabilization. However, the larger size of R-Chord overlay (Theorem 2.3 and Theorem 2.4) increases the stabilization cost ((Theorem 2.5). To amortize the maintenance overhead, periodic stabilization in R-Chord is performed less aggresively, similar to Chord, where each invocation of the stabilization procedure corrects only a subset of a node's fingers, e.g. the successor pointer and another randomly-chosen finger. However, this may increase the time required to correct an R-Chord's overlay.

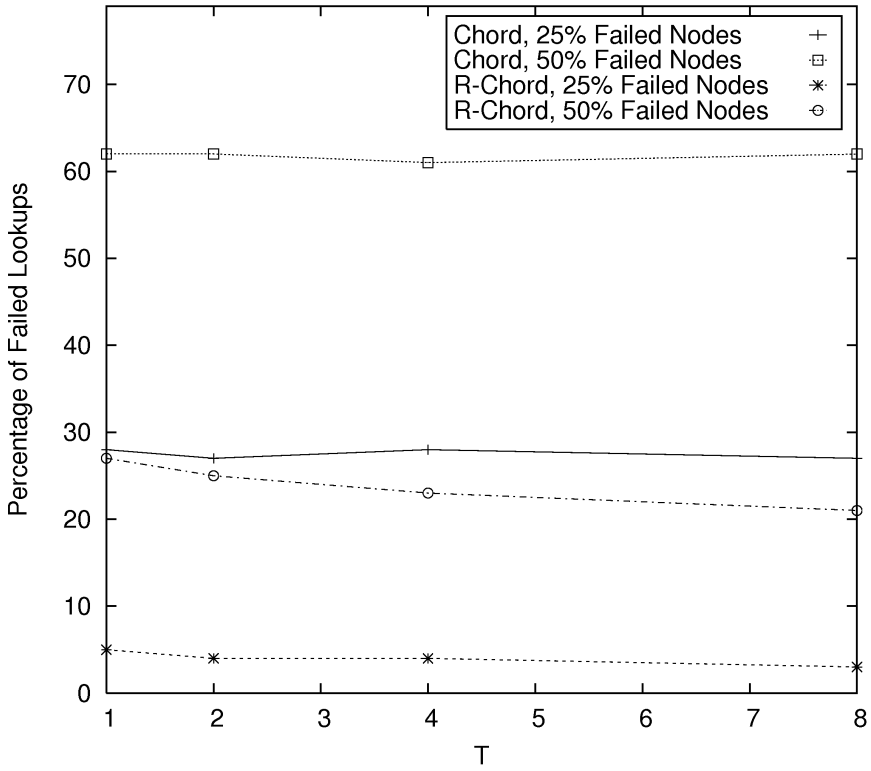


(a) $K = NT$ Unique Keys

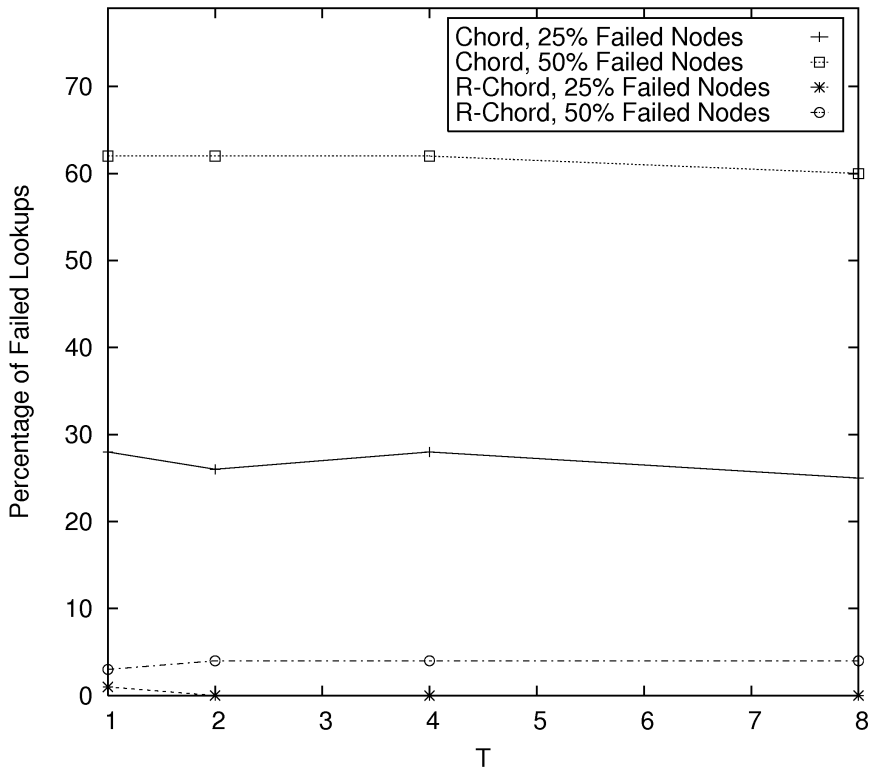


(b) $K = 5,000$ Unique Keys

Figure 2.20: Average Lookup Path Length with Failures (N = 25,000 Hosts)



(a) $K = NT$ Unique Keys



(b) $K = 5,000$ Unique Keys

Figure 2.21: Percentage of Failed Lookups (N = 25,000 Hosts)

In this experiment, we evaluate the time required to correct the overlay topology, which is measured starting from the time when the last host arrives. To facilitate this measurement, we quantify the correctness of an overlay using *stabilization degree* (ξ) which is derived by averaging the correctness of all finger tables in the overlay network (ξ_n).

$$\xi = \frac{\sum_{n=0}^{N-1} \xi_n}{N} \qquad 0 \leq \xi \leq 1 \tag{2.1}$$

and

$$\xi_n = \begin{cases} 0 & \text{if } n.finger[1] \text{ is incorrect} \\ \frac{F'}{F} & \end{cases} \tag{2.2}$$

where $0 \leq \xi_n \leq 1$, F' is the number of correct fingers in n, and F is total number of fingers in n. Note that we do not consider backup fingers in calculating ξ_n.

The experiment is performed as follows. We simulate a system consisting of N hosts with mean arrival rate $\lambda = 1$ host/second. The number of unique keys per host is $|T_h| \sim$ U[2, 5] unique keys and therefore, $T = 3.5$ unique keys. We assume that the total number of unique keys in the system is $K = 3N$ keys; this K approximates NT keys where each segment consists of one node on average. We then periodically measure ξ starting from the time when the last host arrives.

A node joins a ring overlay through a randomly chosen existing node. We base our node-join process on the one described in [133]. First, a new node n starts the join process by adding $n' = \textit{find\_successor}(n)$ as its successor pointer. After one or more rounds of finger correction, there will be at least one other node pointing to n. At this time, the join process completes. Note that R-Chord uses the *find_successor*() which incorporates shared finger tables (Figure 2.18b).

Each node invokes the finger correction every $[0.5p, 1.5p]$ seconds (uniform distribution). Each invocation of finger correction will correct the successor pointer

($n.finger[1]$) and one other finger. Correcting $n.finger[i]$ is done by updating it with the node returned by *find_successor*$(n + 2^{i-1})$.

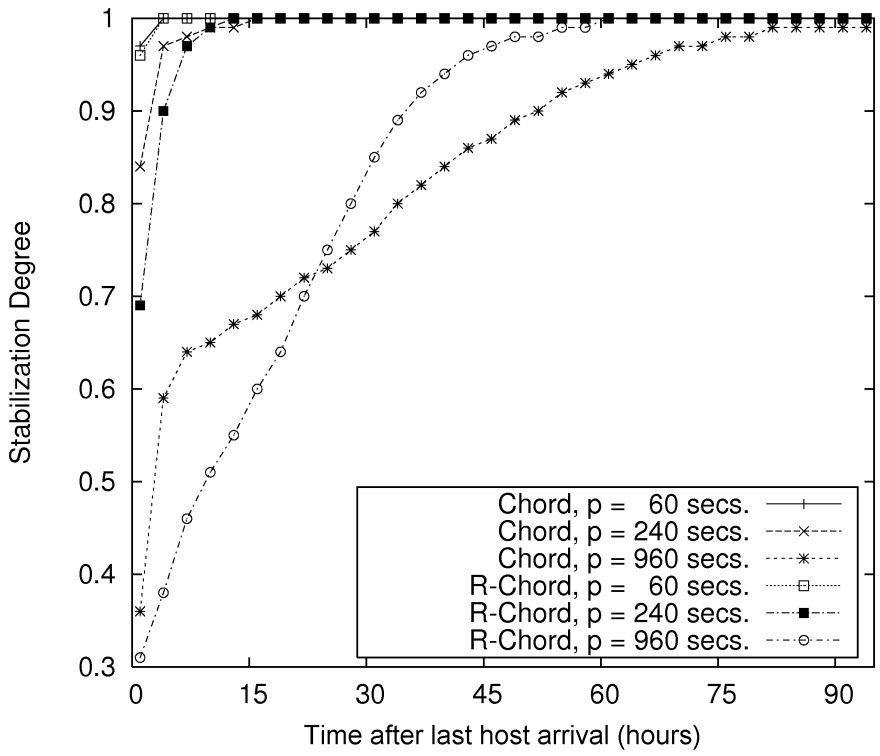
Figure 2.22 reveals that at larger p (960 seconds in this experiment), nodes in Chord correct their successor pointer (i.e. $n.finger[1]$) faster than R-Chord. For examples, when $N = 25,000$ hosts, ξ in Chord increases from 0.36–0.59 in the first three hours, which is faster than R-Chord (0.31 – 0.38). The same behavior is also observed in $N = 50,000$ hosts. This is because ξ_n puts more priority on the successor pointer (see Equation (2.2)). Hence, by correcting successor pointers faster, Chord increases its ξ faster than R-Chord.

Conclusively, although R-Chord has a larger overlay and each of its hosts has to correct more fingers, R-Chord does not require a longer time than Chord to fully correct its overlay. This is due to shared finger tables reducing the time to locate the correct successor when correcting a finger.

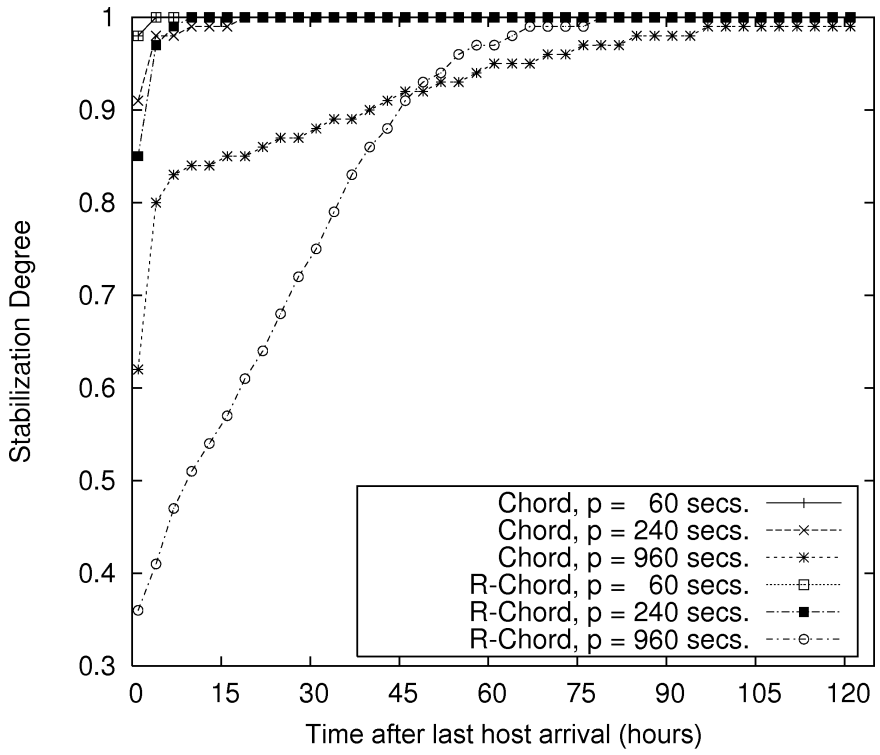
2.5.4 Lookup Performance under Churn

Churn refers to membership changes in an overlay network. In R-DHT, churn occurs in two ways. Firstly, host arrivals, host fails, and host leaves results in simultaneous node joins, node fails, and node leaves, respectively. Secondly, adding a new unique key to a host also causes a node join (Theorem 2.7). When the frequency of membership changes (i.e. *churn rate*) is high, lookup performance may decrease because the larger overlay of R-DHT magnifies the impact of churn on the correctness of overlay topology.

To evaluate the ability of R-DHT to cope with churn, we simulate lookups when R-Chord ring overlay keeps changing due to host arrivals, failures, and leaves. When a node leaves, it notifies its successor and predecessor. In addition, a node leaving



(a) $N = 25,000$ Hosts



(b) $N = 50,000$ Hosts

Figure 2.22: Correctness of Overlay ξ (Measured Every Three Hours Starting From the Last Host Arrival)

a Chord ring migrates all data items belonging to other nodes to its successor without delay.

In this experiment, we assume that each host has $|T_h| \sim \text{U}[4, 12]$ unique keys (which results in $T = 8$ unique keys), and each node invokes the finger correction procedure every $[30, 90]$ seconds (uniform distribution). We first set up an overlay network of 25,000 hosts, followed by a number of churn events (i.e. arrivals, fails, and leaves) produced by 25,000 hosts in a duration of one hour. Thus, there will be about $N = 25,000$ alive hosts at any time within this duration. During this one-hour period, we also simulate a number of lookup events and keep the ratio of arrive:fail:leave:lookup to be 2:1:1:6. The mean arrival rate of these events, λ, will model the churn rate. Assuming that these events follow a Poisson distribution, our simulation uses $\lambda_B = 10$ events/second and $\lambda_G = 40$ events/second; these are derived from the measurements on peer life-time by Bhagwan et. al. [25] and Gummadi et. al. [63], respectively[5]. Table 2.4 presents the result for various K, from 5,000 ($K < N$) to 150,000 ($K \sim NT$).

The average lookup path length (Table 2.4a) again confirms Theorem 2.2 and the result from Subsection 2.5.1. Though the number of nodes in R-Chord's overlay is at least three times Chord, when $K < N$ the average lookup path length is shorter than Chord since R-Chord routes lookups by segments. When $K \geq N$, the average lookup path length is not worse than Chord due to the shared finger tables.

[5] λ_B is obtained as follows. Bhagwan et. al. [25] measures that on average, each host performs 6.4 joins, and 6.4 fails per day [25]. We interpret the measured fail events as consisting of 3.2 host fails and 3.2 host leaves. Thus, including lookups, there are 32 events per day. With 25,000 hosts come and go repeteadly, there are 800,000 events per day, which is approximately one event every 100 ms.
Similar steps as above are used to derive λ_G. Gummadi et. al. [63] measures 24 joins and 24 fails per host per day, and we interpret the measured fails as consisting of 12 host fails and 12 host leaves. Given 25,000 hosts and a ratio of arrive:fail:leave:lookup = 2:1:1:6, there are approximately one event every 25 ms.

K	$\lambda_B = 10$ ev./sec.		$\lambda_G = 40$ ev./sec.	
	Chord	**R-Chord**	**Chord**	**R-Chord**
5,000	8.4	4.1	9.1	4.7
7,500	8.5	4.4	9.1	5.0
10,000	8.5	4.6	9.3	5.4
15,000	8.5	4.9	9.1	5.8
25,000	8.5	5.3	9.2	6.5
50,000	8.5	5.9	9.3	7.2
75,000	8.5	6.2	9.5	7.7
100,000	8.6	6.4	9.4	8.0
125,000	8.6	6.5	9.2	8.1
150,000	8.6	6.6	9.4	8.3

(a) Average Lookup Path Length

K	$\lambda_B = 10$ ev./sec.		$\lambda_G = 40$ ev./sec.	
	Chord	**R-Chord**	**Chord**	**R-Chord**
5,000	2%	<1%	7%	2%
7,500	2%	<1%	9%	2%
10,000	3%	1%	9%	3%
15,000	4%	1%	13%	4%
25,000	4%	1%	15%	9%
50,000	7%	5%	20%	13%
75,000	10%	7%	20%	26%
100,000	7%	5%	23%	25%
125,000	6%	7%	30%	26%
150,000	8%	16%	27%	34%

(b) % of Failed Lookups

Table 2.4: Lookup Performance under Churn ($N \sim 25,000$ Hosts)

Table 2.4b shows that lookup resiliency in R-Chord is comparable to Chord (from 8% lower to 9% higher than Chord). Under a churn rate of λ_B and λ_G, R-Chord has a lower percentage of failed when $K \leq 100,000$ and $K \leq 50,000$, respectively. The results indicate the importance of exploiting finger flexibility through backup fingers. In R-DHT, lookup resiliency is increased due to the property that a key can be found in several nodes of the same segment. Hence, it is important that a segment can be reached as long as it contains one or more alive nodes. R-Chord addresses this issue by maintaining backup fingers as redundant pointers

to a segment. With a higher number of nodes per segment (i.e. finger flexibility), backup fingers are more effective in reducing the impact of a churn rate to lookup resiliency. As K is increased, the number of nodes per segment decreases. Because finger flexibility is reduced, there are less redundancy to be exploited through backup fingers. Considering that R-Chord's overlay is eight times larger than Chord's overlay, we conclude that the decrease is reasonable.

In summary, the result in this subsection suggests that R-Chord achieves better resiliency when finger flexibility is exploited. When R-Chord cannot exploit finger flexibility, it can still achieve comparable resiliency as Chord because by not distributing data items, failure of a host affects only its own data items.

2.6 Related Works

In this section, we first compare and contrast R-DHT with structured P2P systems that support the no-data-item-distribution scheme. Secondly, we discuss the current status of distributed resource indexing and discovery in a computational grid.

2.6.1 Structured P2P with No-Store Scheme

We discuss three structured P2P that also support the no-data-item-distribution scheme, namely SkipGraph [20], Structella [30], and SkipNet [68].

SkipGraph [20] supports the no-store scheme by associating a key to a node and organizing nodes as a skip-list-like topology. It is assumed that each key is shared only by one node, e.g. resources of the same type are shared only by one administrative domain. Our proposed scheme generalizes SkipGraph by first, allowing a key to be associated with several nodes. Secondly, our scheme can organize nodes with different structured overlay topologies.

Structella [30] organizes nodes as a structured overlay (i.e. a Pastry ring [123]) but each node manages only its own keys, similar to R-DHT. However, unlike DHT, Structella does not map keys onto nodes. Instead, Structella employs the routing schemes used in unstructured P2P such as flooding and random walk, and exploits its structured overlay to reduce the overhead of those schemes. The authors reported that Structella offers similar results guarantee as unstructured P2P. To improve results guarantee, the authors propose to distribute and replicate data items to a number of nodes [31]. In contrast, R-DHT maps keys onto nodes and exploits DHT-based lookup schemes. Thus, even without distributing data items, R-DHT offers the same level of result guarantee as other DHT.

SkipNet [68] supports *content locality* to map a key onto a specific node. This is achieved through the hierarchical naming scheme: *put(n|key)* maps a key to node n, and *lookup*($n|key$) retrieves the key. Compared to our proposed scheme, SkipNet provides greater flexibility for a host to decide where its data items are stored. However, the hierarchical naming scheme does not directly supports queries such as "find resources of type k in any hosts". In contrast, though R-DHT addresses node autonomy only by ensuring that data items are stored on their originating host, it is compatible with flat naming scheme.

Table 2.5 summarizes the comparison of R-DHT with the three no-data-item-distribution scheme.

2.6.2 Resource Discovery in Computational Grid

We classify distributed grid information systems based on their overlay topology into unstructured overlay networks [72, 91] and structured overlay networks (DHT) [27, 28, 132, 145].

Characteristic	SkipGraph	Structella	SkipNet	R-DHT
Mapping Scheme	Yes	No	Yes	Yes
High Result Guarantee	Yes	No	Yes	Yes
Key Shared by Many Hosts	No	Yes	Yes	Yes
Controlled Data Placement	No	No	Yes	No
Flat Naming Scheme	Yes	Yes	No	Yes
Overlay Network	Skip List	Pastry	Multi-Level Ring	Multiple Choices

Table 2.5: Comparison of R-DHT with Related Work

In unstructured overlay networks, the routing-transferring model replicates resource information to all nodes proposed [91]. However, this consumes communication bandwidth. Iamnitchi [72] proposes to replicate information based on the small-world effect and uses heuristics to aid lookup. However, heuristics do not guarantee that a lookup will successfully find resources. In contrast, DHT-based systems provides stronger lookup guarantee and scalable lookup performance.

MAAN [28], self-organizing Condor pools [27], XenoSearch [132], and RIC [145] are examples of grid information systems that are based on conventional structured overlay networks. Compared to such schemes, our R-DHT-based grid information system increases the autonomy of administrative domains by not distributing data items. In addition, our scheme does not introduce stale data items when the overlay topology changes, and it is resilient to node failures without a need to replicate data items.

2.7 Summary

Distributed hash table maps each key onto a node to achieve good lookup performance. A typical DHT realizes this mapping through the store operation and as a result, key-value pairs are distributed across the overlay network. To address the requirements of applications where distributing key-value pairs is not desirable, we propose R-DHT, a new DHT mapping scheme without the store operation. R-

DHT enforces the read-only property by virtualizing a host into nodes subjected to the unique keys belonging to the host, and dividing the node identifier space into two sub-spaces (i.e. a key space and a host identifier space). By mapping data items back onto its owner, R-DHT is inherently fault tolerant. In addition, it increases consistency of data items because updates need not be propagated in overlay networks. R-DHT maintains API compatibility with existing DHT. In addition, R-DHT lookup operations exploit not only existing DHT lookup scheme, but also routing by segments, shared finger tables, and finger flexibility through backup fingers.

Through theoretical and simulation analysis, we show that in a Chord-based R-DHT consisting of N hosts and K total unique keys, the lookup path length is $O(\min(\log K, \log N))$ hops. This result suggests that our proposed lookup optimization schemes can reduce lookup path length in other DHTs that also virtualizes hosts into nodes. We further demonstrate that R-DHT lookup is resilient to node failures. In our simulations, when 50% of nodes fail simultaneously, the number of failed lookup in R-Chord is 5–30%; this is lower compared to Chord where at least 60% of its lookups fail. Our simulation also shows that under churn, lookup performance of R-Chord is comparable to Chord even though R-Chord overlay is eight times larger: (i) R-Chord lookup path length is shorter, and (ii) number of failed lookups in R-Chord is at most 8% worst than Chord.

The host-to-nodes virtualization in R-DHT increases the size of overlay network in terms of number of nodes. This leads to higher overhead in maintaining an overlay network. In an R-Chord system that virtualizes N hosts into V nodes where $V \geq N$, the maintenance overhead is $O(V \log V \log N)$. With the same number of hosts, the maintenance overhead in Chord is $O(N \log^2 N)$. Our simulation analysis further revealed that when stabilization period is large, R-Chord requires more

time to correct its overlay into a ring topology. To address the problem of overlay-network maintenance in R-Chord, we present in the next chapter hierarchical R-DHT where nodes are organized into a two-level overlay network.

Chapter 3

Hierarchical R-DHT: Collision Detection and Resolution

Stabilization refers to the procedure for correcting routing states to adapt to changes in an overlay topology. The overlay topology changes when new nodes join, or existing nodes leave or fail. The one-level R-DHT (i.e. flat R-DHT) presented in the previous chapter achieves the same lookup path length as conventional DHT (Theorem 2.1) but with higher stabilization cost, i.e. the maintenance overhead (Theorem 2.5). In a system consisting of N hosts where each host has T unique keys on average, a flat R-DHT overlay consists of V $(= NT)$ nodes. Compared to a conventional DHT overlay consisting of N nodes, R-DHT corrects a higher number routing-table entries which, according to Theorem 2.5, is due to:

1. The number of routing tables is proportional to V, i.e. one routing table per node.

2. The size of each routing table is increased as V increases (Theorem 2.4).

In the case of R-Chord, it takes $\Omega(V \log^2 V)$ stabilization messages to correct all V finger tables, including successor and predecessor pointers, in an R-Chord overlay, i.e. $\Omega(V \log^2 V)$ hops per finger table.

In this chapter, we discuss a hierarchical R-DHT scheme that reduces its maintenance overhead by organizing nodes into a two-level overlay network. To address the problem of *collision of groups*, we propose a collision-detection method that piggybacks on stabilization, and two collision-resolution methods. Collisions occur when concurrent node joins result in nodes with the same *group identifier* being created at the top-level overlay. This increases the size of the top-level overlay, which in turn increases the total number of stabilization messages in the top-level overlay. In the worst case, collisions lead to the degeneration of hierarchical DHT into flat DHT, i.e. every node occupies the top-level overlay.

The rest of this chapter is organized as follows. We first present existing approaches to reduce maintenance overhead in DHT. Next, using R-Chord as the example, we propose a collision detection and resolution scheme. We evaluate our proposed scheme through simulation experiments. Finally, we conclude this chapter with a summary.

3.1 Related Work

A number of approaches have been proposed to reduce the maintenance overhead of DHT. We classify existing approaches into three main categories, (i) varying frequency of stabilization, (ii) varying number of routing states to correct, and (iii) hierarchical DHT. The first two approaches are directly applicable to flat R-DHT.

3.1.1 Varying Frequency of Stabilization

Frequency-based approaches such as adaptive stabilization [29, 52], piggybacking stabilization with lookups [17, 90], and reactive stabilization [17] reduce the maintenance overhead by reducing the frequency in correcting routing states. Adaptive stabilization adjusts the frequency based on churn rate and the importance of each routing state to lookup performance[1]. Systems such as DKS [17] and Accordion [90] piggyback stabilization with lookups to reduce the necessity of performing dedicated periodic stabilization[2]. Reactive stabilization such as DKS's *correction-on-change* [53] does away altogether with periodic stabilization. Instead, changes to overlay networks due to membership changes are propagated immediately when membership-change events are detected. However, Rhea et. al. have reported that reactive stabilization can increase maintenance overhead under high churn rate and constrained bandwidth availability [119].

3.1.2 Varying Size of Routing Tables

This approach reduces the size of routing tables so that the number of routing states to correct becomes smaller. Examples of DHT that implement this approach include CAN [116], Koorde [76], and Accordion [90]. However, reducing the size of routing tables potentially increases lookup path length [139].

Besides reducing the size of routing tables, each routing table can also be partitioned into two parts: one part consisting of entries that are corrected through stabilization, and the other part consisting of cached entries. This reduces the maintenance overhead while achieving a shorter lookup path length. For example, a finger table in Chord consists of $O(\log N)$ fingers and a number of *location caches* where the location caches are maintained by the LRU replacement policy [2].

[1] Routing states with higher importance, e.g. successor pointers in Chord [133] and leaf sets in Pastry [123], are refreshed/corrected more frequently.
[2] In DKS, this is referred as *correction on use*.

3.1.3 Hierarchical DHT

Hierarchical DHT partitions stabilization among different overlays. This speeds up each stabilization process and reduces the number of stabilization messages in each of the overlays. Hierarchical DHT organizes nodes into a multi-level overlay network, where the top-level overlay consists of logical groups [51, 68, 77, 101, 137, 140, 143]. Each group, which consists of a number of nodes, is assigned a *group identifier* based on a common node property. For examples:

1. Grouping by administrative domains [68, 101, 143] improves the administrative autonomy and reduces latency.

2. Grouping by physical proximity [137, 140] reduces network latency.

3. Grouping by services [77] promotes the integration of services into one system.

In each group, one or more *supernodes* act as gateways to the nodes at the second-level. Within each group, nodes can further form a second-level overlay network. In terms of topology maintenance, the hierarchical DHT has the following advantages compared to the flat DHT:

1. Each stabilization message in a hierarchical DHT is routed only in one of the smaller second-level overlays. This reduces the number of stabilization messages processed by each node.

2. Topology changes within a group due to churn do not affect the top-level overlay or other groups. Stable overlay topologies improves the result guarantee of DHT lookups.

In the following, we compare our proposed scheme with existing hierarchical DHT. We discuss how each scheme addresses the problem of collisions.

In hierarchical DHT such as Brocade [143], SkipNet [68], and hierarchical Scribe [101], collisions do not occur because a new node always chooses a *bootstrap node* from the same group. In such systems, nodes are grouped by their administrative domains. Therefore, it is natural for the new node to choose a bootstrap node from the same administrative domain. This grouping policy guarantees that multiple group with the same group identifier are not created. However, such systems do not address other grouping policies that can introduce collisions, i.e. when a new node is bootstrapped from a node in a different group.

In systems such as the hierarchical DHT by Garcés-Erice et. al. [51], Diminished Chord [77], Hieras [140], and HONet [137], collisions can occur but the problem is not directly addressed. They assume that collisions can be resolved by mechanisms inherent in the system structure, and the extent of collisions is not studied.

In [77, 140], all nodes in a group are assumed to be supernodes. In such systems, the size of the top-level overlay, with or without collisions, is the same. Hence, the stabilization procedure of the underlying DHT is sufficient to resolve collisions. However, the size of the top-level overlay is larger than in systems where only a subset of nodes become supernodes. Thus, the total number of stabilization messages is larger because more supernodes have to perform stabilization.

In [51, 137], a new node can choose a bootstrap node from a different group. Hence, it is possible that the bootstrap node cannot locate the group associated with the new node, even though the group exists. However, the effect and impact of the collisions are not evaluated.

To summarize the above comparisons, our scheme relaxes the assumption that a new node must be bootstrapped from the same group and all group members must become supernodes. In addition, our scheme resolves collisions to maintain

the top-level overlay size that is close to the ideal size.

3.2 Design of Hierarchical R-DHT

In an R-DHT framework with V nodes and K ($\leq V$) unique keys, the hierarchical R-DHT organizes the nodes into a two-level overlay network. The top-level overlay consists of K groups, and each group consists of nodes that share the same key. Therefore, groups are equivalent to segments in a flat R-DHT. Every group has one or more supernodes that act as gateways to other nodes in the group. These supernodes are organized in the top-level overlay. Each group can further organize its nodes as a second-level overlay with a topology and stabilization mechanism that differ from the top-level. Clearly, each of the overlay networks in a hierarchical R-DHT is smaller than the flat R-DHT overlay. Thus, while each host h is still virtualized into $|T_h|$ nodes, each of the nodes will join a smaller overlay network than in the flat R-DHT network. As a result, each node maintains and corrects a smaller number of fingers than the flat R-DHT's nodes. Figure 3.1 shows a hierarchical R-Chord where the top-level ring consists of four groups.

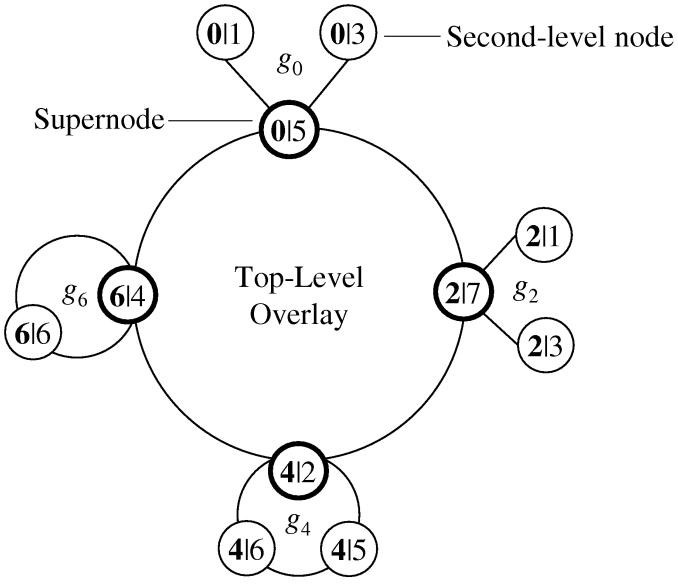


Figure 3.1: Two-Level Overlay Consisting of Four Groups

Nodes in the hierarchical R-DHT are assigned two identifers, as opposed to nodes in the flat R-DHT. In the flat R-DHT, $n = k|h$ denotes that $k|h$ is the node identifier of node n. In the hierarchical R-DHT, we also assign a group identifier to node n. The value of the group identifier is equal to $prefix(n)$, which is k. In addition, each second-level node in the hierarchical R-DHT holds a pointer to at least one of the supernodes in its group. Table 3.1 summarizes the important variables maintained by each node, in addition to the ones presented in Figure 2.1.

Variable	**Description**
gid	m-bit group identifier ($= prefix(n)$)
is_super	**true** if n is a supernode, **false** otherwise
supernode	Pointer to supernode of group *gid*, **nil** if n is a supernode.

Table 3.1: Additional Variables Maintained by Node n in a Hierarchical R-DHT

In hierarchical R-DHT, locating key k implies locating the group responsible for k. Firstly, a lookup request for key k is routed to the supernode of the initiating group. Secondly, using R-DHT lookup algorithm (Figure 2.13), the lookup request is further routed to the supernode of group k, i.e. a supernode whose node identifier is prefixed with k. Thirdly, the lookup request can be further forwarded to one of the second-level nodes in group k, depending on the application. As illustrated in Figure 3.2, a lookup request for key 2, initiated by second-level node 6|6, is forwarded to its supernode 6|4 (step 1). In the top-level overlay, the lookup request is routed to supernode 2|7 of group 2 (step 2). Finally, supernode 2|7 can further forward the request to its second-level nodes (step 3), e.g. lookup for compute resources of type 2 in multiple administrative domains.

If new nodes join hierarchical R-DHT when some routing states in the top-level overlay are incorrect, i.e. yet to be updated, the top-level overlay may end up with two or more groups with the same group identifier. In the following subsections,

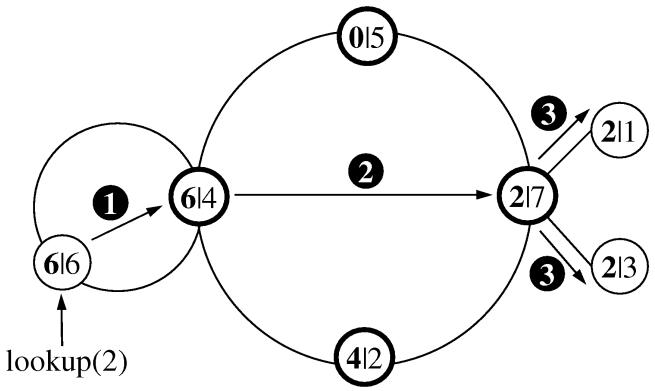


Figure 3.2: Example of a Lookup in Hierarchical R-DHT

we discuss how collisions occur, and then present our proposed scheme to detect and resolve collisions. To avoid sending additional overhead messages, collision detection is performed together with *successor stabilization*, i.e. the process of correcting successor pointers. This is because successful collision detections require the successor pointers in the top-level Chord overlay to be correct, and the correctness of the successor pointers is maintained by stabilization.

3.2.1 Collisions of Group Identifiers

Collisions of group identifiers arise because of join operations invoked by *nodes*. Figure 3.3 shows the node-join algorithm for hierarchical R-Chord. Node n, whose group identifier is denoted as $n.gid$, makes a request to join group g through bootstrap node n'. In a hierarchical R-Chord, this means finding $successor(g|0)$ in the top-level overlay. If n' successfully finds an existing group g, then n joins this group using a group-specific protocol (line 5–9). However, if n' returns $g' > g$, then n creates a new group with identifier g (line 11–15). A collision occurs if the new group is created even though a group with identifier g has already been created. This happens due to n and bootstrap node n' are in two different groups, and the top-level overlay has not fully stabilized (i.e. some supernodes have incorrect successor pointers).

```
1.   // Node n joins through bootstrap node n'
2.   n.join(n')
3.       h' = suffix(n');
4.       s = h'.find_successor(gid|0);  // See Figure 2.18b
5.       if (gid == s.gid)
6.           // s is a supernode of group g
7.           join_group(s);
8.           is_super = false;
9.           supernode = s
10.      else
11.          // n creates a new group.
12.          // This can cause a collision.
13.          predecessor = nil;
14.          successor = s;
15.          is_super = true;
```

Figure 3.3: Join Operation

Figure 3.4 illustrates a collision that occur when node 1|2 and node 1|3 belonging to the same group g_1, join concurrently. Due to concurrent joins, *find_successor*() invoked by both nodes, during their join operation, will return node 2|7. This causes both the new nodes to create two groups with the same group identifier g_1.

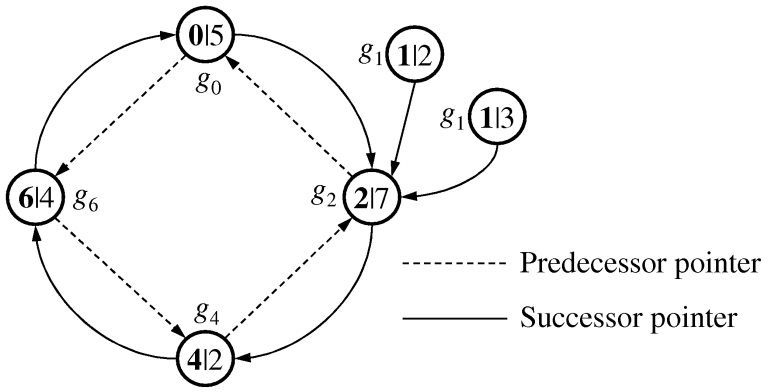


Figure 3.4: Collision at the Top-Level Overlay

3.2.2 Collision Detection

We propose to perform collision detections during successor stabilization. This is achieved by extending Chord's stabilization so that it not only checks and corrects

the successor pointer of supernode n, but also detects if n and its new successor should be in the same group. Figure 3.5 presents our collision detection algorithm, assuming that each group has only one supernode. The algorithm first ensures that the successor pointer of a node is valid (line 4–5). It then checks for a potential collision (line 8–10), before updating the successor pointer to point to the correct node (line 11–13).

```
1.   // n periodically verifies its successor pointer,
2.   // and announces itself to the successor.
3.   n.stabilize_successor()
4.       if successor.is_super == false then
5.           successor = successor.supernode();
6.
7.       p = successor.predecessor;
8.       if ((p ≠ n) and (p.gid == gid)) then
9.           if is_collision(p) then
10.              merge(p);
11.      else if n.gid < p.gid < successor.gid then
12.          successor = p;
13.          successor.notify(n);
```

(a) Main Algorithm

```
14.  // n' thinks it might be our predecessor
15.  n.notify(n')
16.      if (predecessor == nil)
17.          or (predecessor.is_super == false)
18.          or (predecessor < n' < n)
19.        then
20.          predecessor = n';
```

```
21.  // Assume one supernode per group
22.  n.is_collision(n')
23.      if (gid == n'.gid)
24.          return true
25.
26.      return false
```

(b) Helper Functions

Figure 3.5: Collision Detection Algorithm

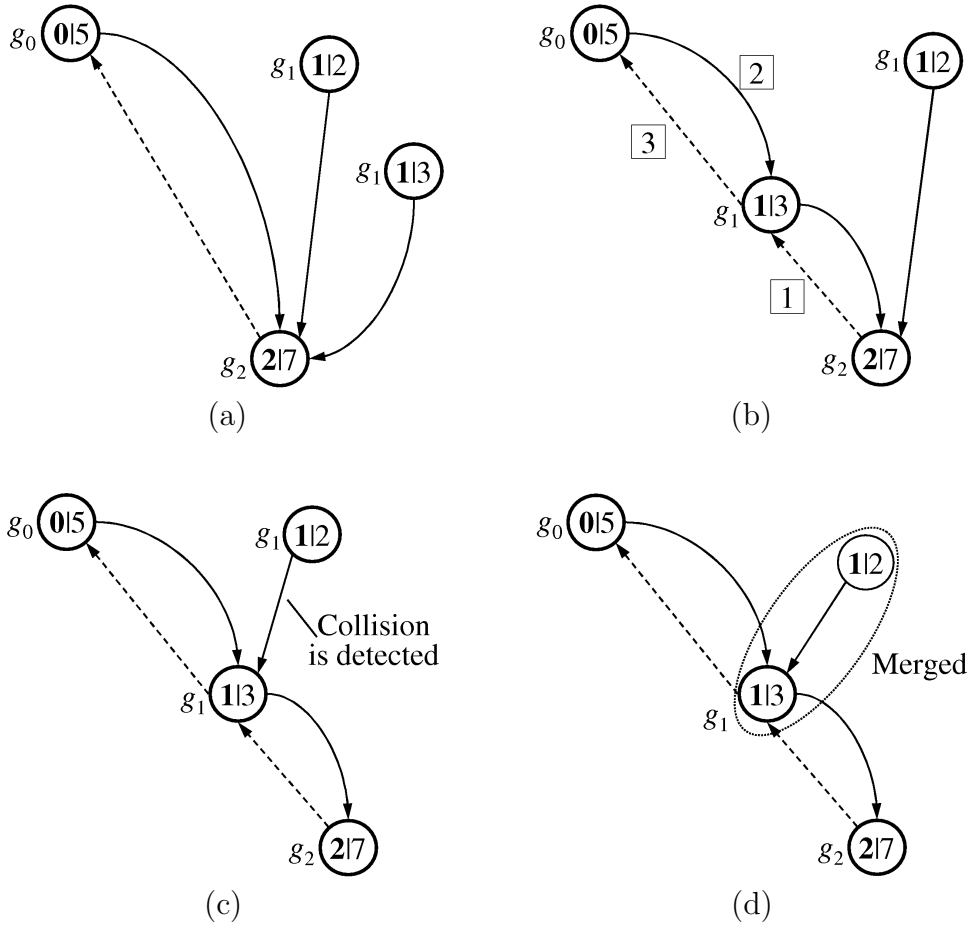
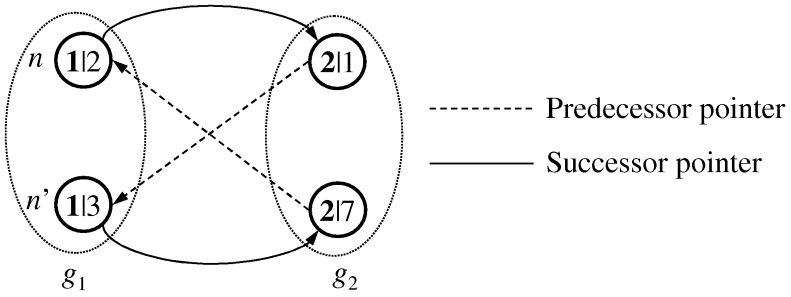


Figure 3.6: Collision Detection Piggybacks Successor Stabilization

The following example illustrates the collision detection process. In Figure 3.6a, a collision occurres when node 1|2 and 1|3 belonging to the same group, group 1, join concurrently. In Figure 3.6b, node 1|3 stabilizes and causes node 2|7 to set its predecessor pointer to node 1|3 (step 1). Then, the stabilization by node 0|5 causes 0|5 to set its successor pointer to node 1|3 (step 2), and node 1|3 to set its predecessor pointer to node 0|5 (step 3). In Figure 3.6c, the stabilization by node 1|2 causes 1|2 to set its successor pointer to node 1|3. At this time, a collision is detected by node 1|2 and is resolved by merging 1|2 to 1|3.

If each group contains more than one supernodes, then *is_collision* routine shown in Figure 3.5 may incorrectly detect collisions. Consider the example in Fig-

ure 3.7a. When node n stabilizes, it incorrectly detects a collision with node n' because $n.successor.predecessor = n'$ and $n.gid = n'.gid$. An approach to avoid this problem is for each group to maintain a set of its supernodes [51, 65] so that each supernode can accurately decide whether a collision has occurred. The modified collision detection algorithm is shown in Figure 3.7b.



(a) Multiple Supernodes in Each Group

```
1.  n.is_collision(n')
2.      // L is a set of supernodes in my group
3.      if n' ∉ L then
4.          return true
5.
6.      return false
```

(b) Modified *is_collision* Algorithm

Figure 3.7: Collision Detection for Groups with Several Supernodes

3.2.3 Collision Resolution

To resolve collisions, groups associated with the same *gid* are merged. After the merging, some supernodes become ordinary nodes depending on the group policy. Before a supernode changes its state into a second-level node, the supernode notifies its successors and predecessors to update their pointers (Figure 3.8). Nodes in the second level also need to be merged to the new group. We propose two methods to merge groups, namely *supernode initiated* and *node initiated*.

```
1.  // Set predecessor of n to n'
2.  n.replace_predecessor(n')
3.      predecessor = n';
4.  // Set successor of n to n'
5.  n.replace_successor(n')
6.      successor = n';
```

Figure 3.8: Announce Leave to Preceding and Succeeding Supernodes

```
1.  // Nodes joins the group where n' is the supernode
2.  n.merge(n')
3.      is_super = false
4.
5.      // Announce leave to neighbors in top-level overlay
6.      successor.replace_predecessor(predecessor);
7.      predecessor.replace_successor(successor);
8.      predecessor = successor = nil;
9.
10.     n'.join_group(n);
11.     g = prefix(n);
12.     for each node x ∈ g do
13.         x.join_group(n');
14.         x.supernode = n'
```

Figure 3.9: Supernode-Initiated Algorithm

3.2.3.1 Supernode Initiated

To merge a group $n.gid$ with another group $n'.gid$, the supernode n notifies its second-level nodes to join group $n'.gid$ (Figure 3.9). The advantage of this approach is that second-level nodes join a new group as soon as a collision is detected. However, n needs to keep track of its group membership, which may not always be correct. If n has only partial knowledge of group membership, some nodes in the second-level can become orphans.

3.2.3.2 Node Initiated

In node-initiated merging, each second-level node periodically checks that its known supernode n' is still a supernode (Figure 3.10). If n' is no longer a su-

pernode, then the second-level node will ask n' to find the correct supernode and join a new group through the new supernode. This approach does not require supernodes to track group membership. However, an additional overhead is introduced due to second-level nodes periodically checking the status of their supernode.

```
1.  // Supernode n joins another group,
2.  // ignoring its second-level nodes
3.  n.merge(n')
4.      is_super = false;
5.
6.      // Announce leave to neighbors in top-level overlay
7.      successor.replace_predecessor(predecessor);
8.      predecessor.replace_successor(successor);
9.      predecessor = successor = nil;
```

(a) Main Algorithm

```
10.  // Second-level node n periodically
11.  // verifies its supernode pointer
12.  n.check_supernode()
13.      if supernode.is_super == false then
14.          x = supernode.supernode;
15.          supernode = x;
16.          join_group(x);
```

(b) Helper Functions

Figure 3.10: Node-Initiated Algorithm

3.3 Simulation Analysis

To evaluate the effectiveness of our proposed scheme, we first show that hierarchical R-Chord significantly reduces maintenance overhead, compared to flat R-Chord. Then, we study the performance of collision detection and resolution by comparing two systems: *without detect & resolve* (i.e. hierarchical R-Chord without collision detection and resolution) and *detect & resolve* (i.e. hierarchical R-Chord with collision detection and resolution). We assume that each group

contains one supernode. To resolve collisions, we use the supernode-initiated approach. Since the emphasis of this experiment is to study collisions at the top-level R-Chord and the purpose of collision resolution is to ensure that second-level overlays are correct after a collision, the choice of collision-resolution approach does not significantly affect the result of this experiment.

We extend the Chord simulator included in Chord SDK [2] to model a hierarchical R-Chord. The average inter-arrival time of nodes is exponentially distributed with a mean of one second. Each supernode maintains a successor pointer, a predecessor pointer, and $O(\log\ K_C)$ fingers. In addition, each supernode periodically invokes the stabilization procedure. With a stabilization period parameter of p (in seconds), the stabilization period is uniformly distributed in the interval $[0.5p, 1.5p]$. In the simulators, each stabilization corrects the successor pointer and one of the fingers. The link latency between nodes is exponentially distributed with a mean of 50 ms and the request-processing time by each node is uniformly distributed between 5 and 15 ms.

3.3.1 Maintenance Overhead

We measure the maintenance overhead by the total number of stabilization messages in the top-level overlay. We simulated hierarchical and flat systems with 50,000 and 100,000 nodes (V). For the number of groups (K) in the top-level overlay, we chose the values of 2,000 and 8,000. Thus, we evaluated four different R-Chord configurations. In addition, we compare the maintenance overhead of hierarchical R-Chord to Chord. The results are shown in Figure 3.11.

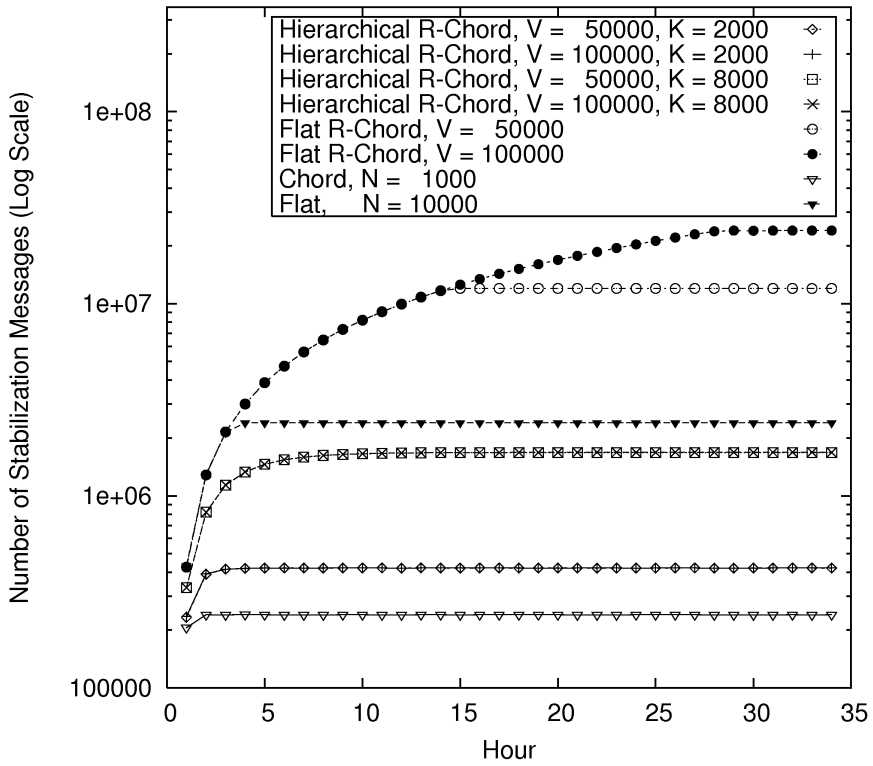
As shown in Figure 3.11, hierarchical R-Chord significantly reduces the maintenance overhead of its top-level overlay compared to flat R-DHT, because the top-level overlay consists of only K groups. Hence, there are a smaller number of

fingers to correct. When we double V from 50,000 to 100,000 nodes, the total number of stabilization messages does not increase; this is in contrast to flat R-Chord. In both systems, the number of stabilization messages reduces by 50% when p is increased from 30 seconds to 60 seconds, because stabilization is performed less frequently.

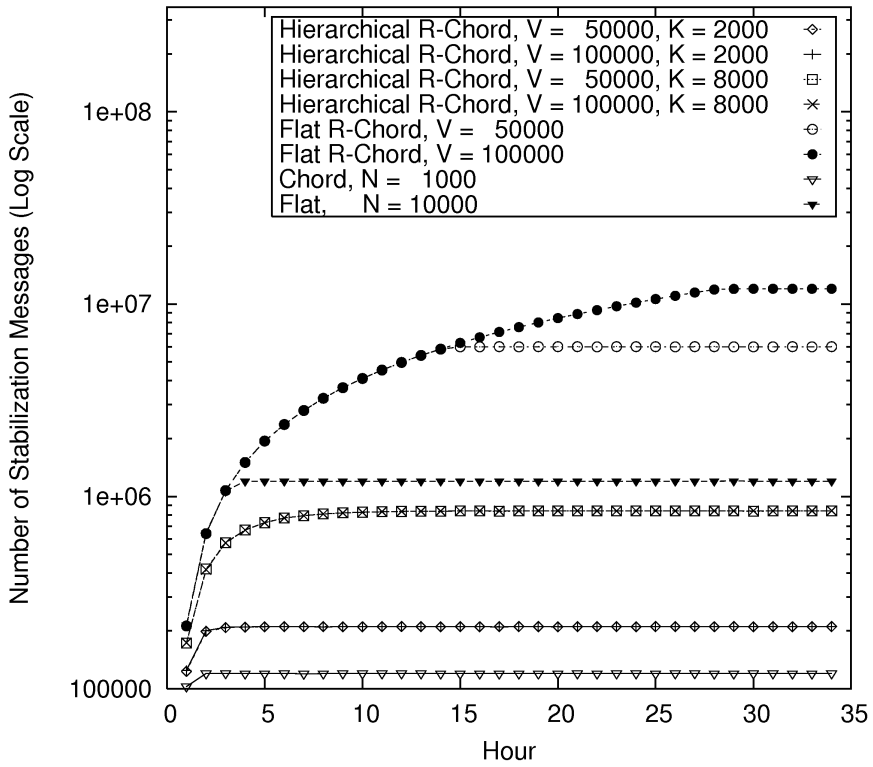
The overhead to maintain the top-level overlay of hierarchical R-Chord is lower than Chord consisting of $N = 10,000$ hosts, but is still higher than Chord consisting of $N = 1,000$ hosts. This is because the maintenance overhead of hierarchical R-Chord and Chord depends on K and N, respectively. In hierarchical R-Chord, the maintenance overhead in the top-level overlay is $O(K \log^2 K)$ because there are K groups in the top-level overlay and each supernode of a group maintains $O(\log K)$ fingers[3]. Note that the total cost to maintain a hierarchical R-Chord, which includes the cost to maintain second-level overlays, is $\Omega(V \log N \log V)$. However, this can be amortized through less frequent stabilizations in the second-level overlays. Unlike the top-level overlay where an incorrect topology causes all nodes sharing the same resource type to be inaccessible, an incorrect second-level overlay causes only a subset of nodes sharing the same resource type to be inaccessible.

We discuss briefly the comparison with the hierarchical DHT schemes presented in Section 3.1.3. In [51, 68, 101, 137, 143], the maintenance overhead for their top-level overlay is higher than hierarchical R-DHT when their top-level overlay is larger than K. In [77, 140], the size of their top-level overlay is always $O(N)$. In addition, each node joins more than one overlay network, i.e. the top-level overlay plus the lower-level overlay networks. Hence, the maintenance overhead for their top-level overlay is higher than hierarchical R-DHT when $N > K$. Moreover, in the case of Hieras [140], the number of overlay levels can be greater than two and

[3]The proof is similar as in Theorem 2.5.



(a) $p = 30$ Seconds



(b) $p = 60$ Seconds

Figure 3.11: Maintenance Overhead of Hierarchical R-Chord

each node is present in every level. Therefore, the maintenance overhead of the whole Hieras overlays is higher when the size of its all overlays is greater than V.

3.3.2 Extent and Impact of Collisions

Consider the total number of stabilization messages required at the top-level R-Chord overlay. Let K ($\leq N$) denote the number of groups and V denote the number of nodes. Each group employs one supernode and hence, we expect that the ideal size of the top-level overlay consists of K supernodes. *Without collisions*, the total number of stabilization messages (S) is $O(K \log^2 K)$ because there are K groups that perform stabilization, each group corrects $O(\log\ K)$ fingers, and the cost of correcting each finger is $O(\log\ K)$. *With collisions*, the size of top-level overlay is increased by c times, i.e. cK groups. As each group performs periodic stabilization, the cost of stabilization when collisions occur (S_C) is $\Omega(cS)$ (Equation 3.1).

$$\frac{S_c}{S} = \frac{cK \log^2 cK}{K \log^2 K} = \frac{c \log^2 cK}{\log^2 K} = \Omega(c) \tag{3.1}$$

Table 3.2 shows the extent of collisions from measuring the total number of collisions for different values of the stabilization period p. Without resolving collisions, the number of collisions is about 2 to 5 times K. With frequent stabilization, our scheme significantly reduces the number of collisions. But as p increases, the number of collisions grows because of the reduced frequency of collision resolution.

The impact of collisions is measured by the growth in the size of the top-level overlay. Figure 3.12 shows the number of groups at an interval of one hour. Without collision resolution, the size of the top-level overlay grows to about 2 to 5 times K because the additional groups caused by collisions will remain in the

p	***Without Detect & Resolve***		***Detect & Resolve***	
	$K = 2,000$	$K = 8,000$	$K = 2,000$	$K = 8,000$
30	5,740	11,421	56	33
60	5,941	11,511	113	153
120	6,425	12,823	1,181	1,088
240	8,914	15,905	1,609	2,349

(a) V = 50,000 Nodes

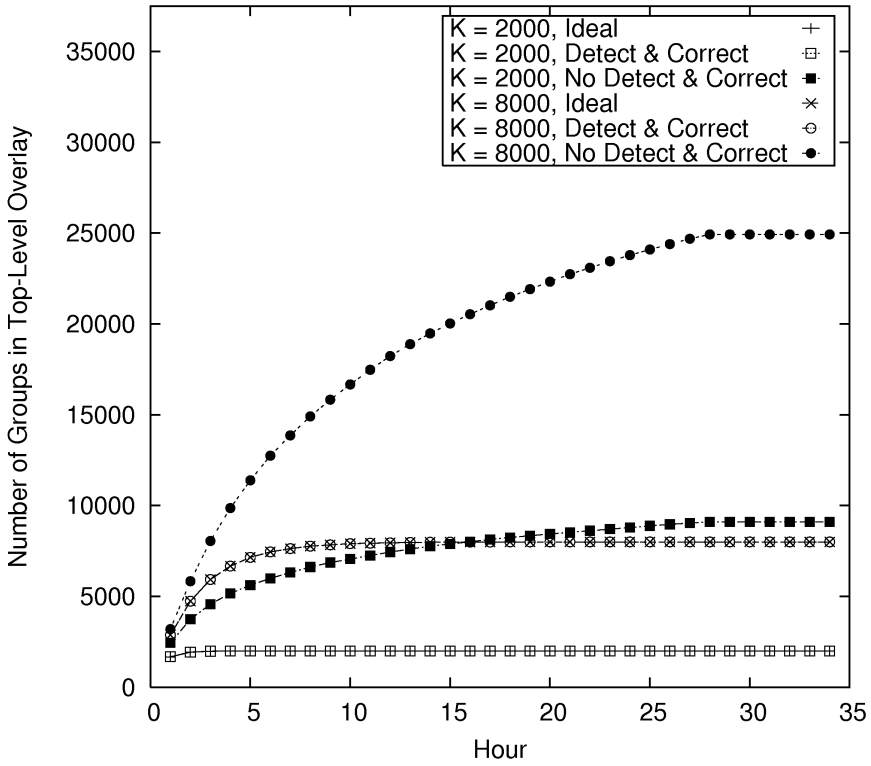
p	***Without Detect & Resolve***		***Detect & Resolve***	
	$K = 2,000$	$K = 8,000$	$K = 2,000$	$K = 8,000$
30	7,097	16,930	35	23
60	7,232	17,009	212	136
120	7,830	17,979	641	1,133
240	9,813	20,139	1,942	3,023

(b) V = 100,000 Nodes

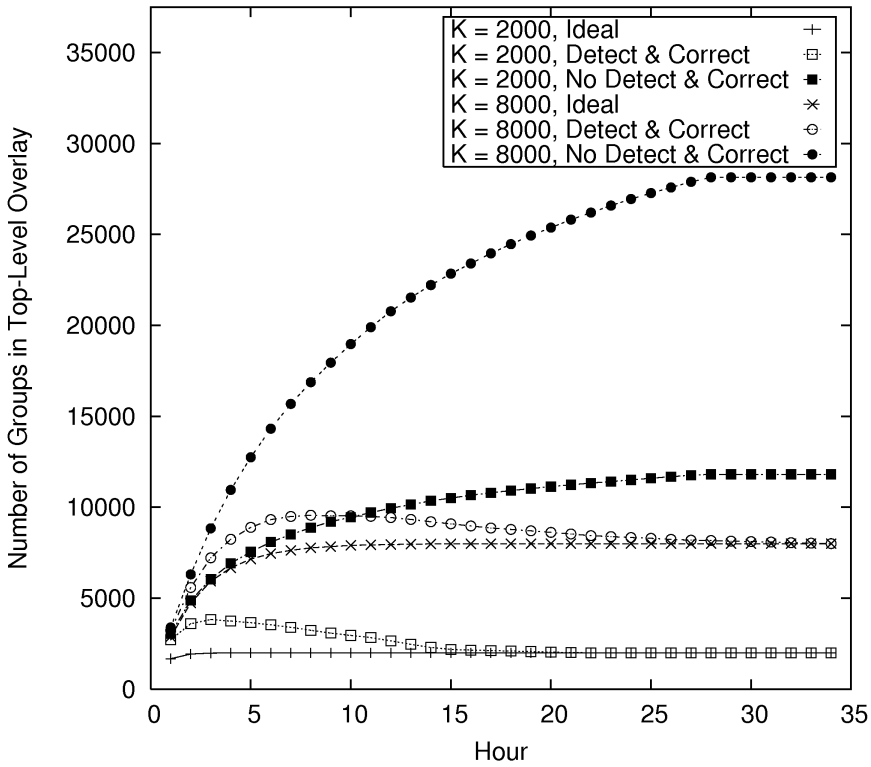
Table 3.2: Number of Collisions

top-level overlay. If the size of the top-level overlay increases by 5 times, then the total number of stabilization messages is increased by $\Omega(5)$ times. On the other hand, *detect & resolve* merges the colliding groups so that the size of the overlay converges to that of the ideal size K.

Figure 3.12 also shows that more frequent stabilization keeps the top-level overlay size that is close to the ideal size. With a larger p, stabilization is performed less frequently. Thus, more stabilization rounds are required to correct the successor pointers. Since our scheme is performed together with stabilization to reduce overhead, it takes a longer time to reduce the size of the top-level overlay close to the ideal size. As an example, with $p = 240$ seconds, it takes at least 15 hours to reduce the top-level overlay size to the ideal size (Figure 3.12b).



(a) $p = 30$ Seconds



(b) $p = 240$ Seconds

Figure 3.12: Size of Top-Level Overlay ($V = 100,000$ Nodes)

3.3.3 Efficiency and Effectiveness

The efficiency and effectiveness of our scheme depends on the frequency of detection and resolution, which is determined by the stabilization period p.

3.3.3.1 Detection

The efficiency of collision detection is measured by the average time required to detect a collision. This is defined as the period between a join and a stabilization procedure that detects the collision. It is desirable to detect collisions as soon as possible to minimize the impact of collisions. Table 3.3 shows that the average time to detect collisions increases as p increases. From the results, the ratio of the collision detection time to the stabilization interval (p) is up to 104 times (i.e. $p = 120$ seconds and $K = 8,000$). This indicates that collision detection time is significant.

p	$V = 50,000$		$V = 100,000$	
	$K = 2,000$	$K = 8,000$	$K = 2,000$	$K = 8,000$
30	1,265	186	288	61
60	3,211	2,849	1,764	4,236
120	5,955	9,635	5,557	12,526
240	9,281	22,070	6,960	23,646

Table 3.3: Average Time to Detect a Collision (in Seconds)

Table 3.4 shows the effectiveness of our scheme. Let β denote the ratio of the number of collisions in the *detect & resolve* case to the number of collisions in the *without detect & resolve* case. With frequent stabilization when p is 30 seconds, β is less than 0.01, i.e. our scheme reduces the number of collisions by 99%. As p increases, the effectiveness of the scheme decreases. However, even when p is 240 seconds, our scheme still reduces the number of collisions by at least 80%.

p	$V = 50,000$		$V = 100,000$	
	$K = 2,000$	$K = 8,000$	$K = 2,000$	$K = 8,000$
30	0.01	0.01	0.01	0.01
60	0.02	0.01	0.02	0.03
120	0.18	0.08	0.08	0.11
240	0.18	0.15	0.13	0.20

Table 3.4: Ratio of Number of Collisions (β)

3.3.3.2 Resolution

There are two main factors that affect the cost of collision resolutions. The first factor is the number of groups and nodes to be merged. Table 3.5 shows the average number of nodes corrected in each collision resolution. The effectiveness of collision resolution improves with a higher frequency of stabilization. Overall, our results indicate that the average number of nodes corrected can be reduced to less than 10% of the average group size (V/K).

p	$V = 50,000$		$V = 100,000$	
	$K = 2,000$	$K = 8,000$	$K = 2,000$	$K = 8,000$
30	2.2	2.1	3.2	2.1
60	2.9	2.5	3.1	2.3
120	3.6	4.0	7.2	3.6
240	7.0	4.9	11.8	6.2

Table 3.5: Average Number of Nodes Affected by a Collision

The second factor is the overhead of correcting stale finger pointers and the cost of updating fingers to point to the new group after merging. As each group is pointed by $O(\log\ K_C)$ groups and the correction of each finger pointer requires $O(\log\ K_C)$, the total cost to update the fingers pointing to the merged group is $O(\log^2\ K_C)$.

The results in this section, i.e Tables 3.3–3.5, suggest that the efficiency and

effectiveness of our scheme can be improved by having more frequent detections and resolutions. This will reduce both the number of collisions and the cost of correcting collisions. Based on the simulation results in Table 3.2, with $p = 60$ seconds, the number of collisions is smaller than 12% of the ideal size (when $V = 100,000$ and $K = 2,000$).

3.4 Summary

We have presented a hierarchical R-DHT and a scheme to detect and resolve collision of groups. A hierarchical R-DHT organizes nodes into a two-level overlay network. It partitions stabilization among different overlays to speed-up each stabilization process and reduces the number of stabilization messages in each overlay. In the hierarchical R-DHT, the maintenance overhead of the top-level overlay is $O(K \log^2 K)$. However, collision of groups increases the size of the top-level overlay by a factor c, which increases the total number of stabilization messages by $\Omega(c)$ times. Our scheme performs collision detection together with stabilization to avoid introducing additional messages. Two approaches are proposed to resolve collisions: supernode-initiated merging and node-initiated merging.

Our simulation results show that if collisions are not resolved, the size of the top-level overlay increases more than twice. With our scheme, the number of collisions is reduced by 80% at least. In addition, the size of the top-level overlay remains close to the ideal size; otherwise it can be up to five times larger, which increases the total number of stabilization messages by $\Omega(5)$ times. The results also reveal the importance of minimizing collisions as it takes several stabilization rounds to detect collisions. Thus, more frequent stabilization reduces collisions and keeps the top-level overlay that is close to the ideal size.

Chapter 4

Midas: Multi-Attribute Range Queries

DHT supports lookup with exact queries effectively (i.e. high result guarantee) and efficiently (i.e. short lookup path length). An exact query locates resources identified with a specific key. As an example, a query *find files whose* `file_name` = *A.MP3* locates all files identified with a key of SHA1(A.MP3). Recently, supporting efficient multi-attribute range queries on DHT has been an active area of research (see Section 1.3). A multi-attribute query locates resources identified with multiple search attributes. Each search attribute can be constrained by a range of values using relational operators ($<$, $\leq$, $=$, $>$, and $\leq$). As an example, *find compute resources whose* `cpu` = *P3 and 1 GB* $\leq$ `memory` $\leq$ *2 GB* is a query consisting of two search attributes; the second search attribute, `memory`, has a range of *1 GB*.

We propose Midas (***M***ult***i***-***d***imensional r***a***nge querie***s***), an approach to support multi-attribute range queries on R-DHT based on *d*-to-*one* mapping scheme. We

focus on resources that are described by a well-defined schema, e.g. GLUE for describing compute resources [5]. Midas adopts the Hilbert space-filling curve (Hilbert SFC) [124] as the *d*-to-*one* mapping function because it has been shown that for multi-dimensional indexing, it has a better clustering property than other types of SFC [74, 103].

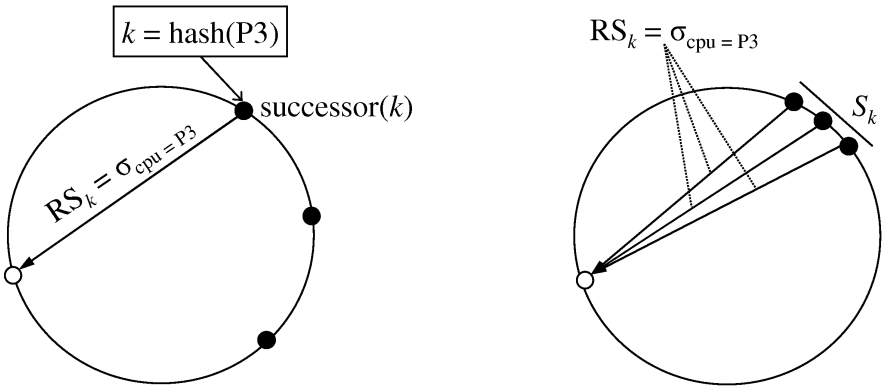
The rest of this chapter is organized as follows. First, we discuss the related work, followed by an overview of Hilbert space-filling curve. Next, we discuss the design of Midas indexing scheme, followed by two optimizations of the the query engine, namely *incremental search* and *search-key elimination*. The performance of Midas is evaluated using simulations. Finally, we conclude this chapter with a summary.

4.1 Related Work

We compare Midas with three main approaches in supporting multi-attribute range queries on DHT, namely distributed inverted index, *d*-to-*d* mapping, and *d*-to-*one* mapping (Section 1.3). We outline the rationale for choosing *d*-to-*one* as the basis for supporting multi-attribute range queries in R-DHT.

Compared to distributed inverted index, *d*-to-*one* mapping does not need to perform the intersection operator ($\cap$). The intersection operation assumes that one or more intermediate results are created using selection operators (σ). However, a selection operation incurs a higher overhead in R-DHT as it visits every node within an R-DHT segment for creating an intermediate result. With *d*-to-*one*, Midas avoids the intersection and thus, does not need to create intermediate result sets. Figure 4.1 compares how Chord and R-Chord create an intermediate result set consisting of resources whose `cpu` = *P3*. In Chord, resources whose `cpu` = *P3* are indexed by a key of $k = hash(P3)$. Thus, the select operation retrieves the relevant indexes from $successor(k)$ only. However, the same operation in R-Chord

retrieves the relevant indexes from all nodes in segment S_k.



(a) Chord: Retrieve Key k from $successor(k)$

(b) R-Chord: Retrieve Key k from All Nodes in Segment S_k

Figure 4.1: Retrieving Result Set of Resource Indexes with Attribute cpu $= P3$

Compared to d-to-d mapping scheme, d-to-*one* mapping offers a higher flexibility in selecting the underlying DHT. Because resources are mapped onto keys in a one-dimensional identifier space, we can use one of the many implementations of one-dimensional DHT [14, 17, 20, 68, 76, 99, 119, 122, 123, 133, 144] as the underlying infrastructure. On the other hand, d-to-d mapping requires multi-dimensional DHT which, to the best of our knowledge, is implemented only by CAN [116].

A number of d-to-*one* mapping schemes have been proposed for DHT [16, 50, 86, 127, 131]. These schemes reduce the number of nodes visited during query processing by exploiting data-item distribution. Though Midas is also based on d-to-*one* mapping scheme, it is designed for R-DHT which does not distribute data items. To reduce query cost, Midas transforms a query into a number of search keys and performs R-DHT lookups only for available keys.

Table 4.1 summarizes the comparison of the three query processing schemes.

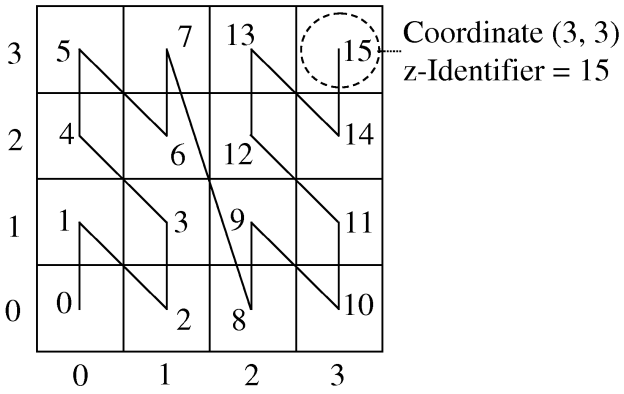
Factor	**Distributed Inverted Index**	***d*-to-*d***	***d*-to-*one***	
			General	**R-DHT**
#keys/resource	d	one	one	one
Type of DHT	any	d-dimensional	any	any
Query engine	$\cap$ and σ	flood query region	exploit data-item distribution	search-key elimination

Table 4.1: Comparison of Multi-attribute Range Query Processing

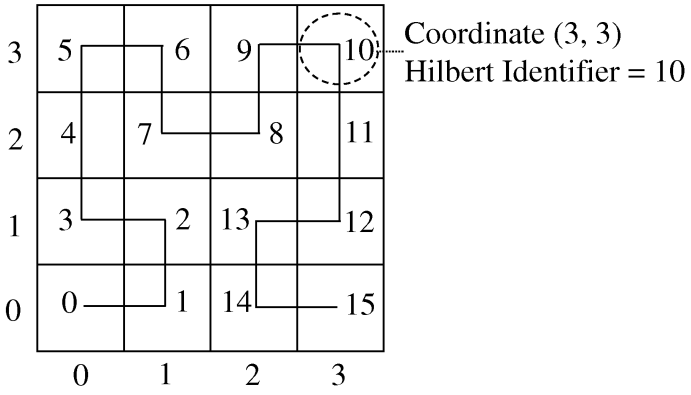
4.2 Hilbert Space-Filling Curve

Let $f : \mathbb{N}^d \rightarrow \mathbb{N}$ denote a d-to-*one* mapping function which maps a d-dimensional space to a one-dimensional space. The function is also referred to as a space-filling curve (SFC) because it can be visualized as a curve (i.e. the one-dimensional space) that traverses every coordinate in the d-dimensional space. A coordinate in a d-dimensional space is a tuple of d *dimension values*. Figure 4.2 illustrates two types of SFC, namely z-curve and Hilbert curve, on a 2-dimensional space consisting of two axes, x-axis and y-axis. Each dimension consists of four values from 0 to 3, resulting in a total of 16 coordinates (cells). SFC has been used in various applications, including multi-dimensional indexing in traditional databases [9, 45, 85, 115].

SFC allows every coordinate in a d-dimensional space to be assigned a unique identifier. The curve is divided into subcurves such that a coordinate covered by the i^{th} subcurve, where $i > 0$, is assigned identifier $i - 1$. The curve traverses the whole d-dimensional space where every coordinate is covered by one subcurve only; this ensures all coordinates are assigned a unique identifier. Figure 4.2 shows that coordinate (0, 0) has the same z-identifier and Hilbert identifier, which is 0, as it is covered by the first subcurve of both SFC. On the other hand, coordinate (3, 3) has two different identifiers: 15 and 10 as its z-identifier and Hilbert identifier,



(a) z-Curve



(b) Hilbert Curve

Figure 4.2: SFC on 2-Dimensional Space

respectively.

In Chapter 4.2.1–4.2.2, we present the locality property of Hilbert SFC and the recursive construction of a Hilbert curve.

4.2.1 Locality Property

An SFC preserves *locality* if coordinates close in the d-dimensional space are mapped onto identifiers close in the one-dimensional space, and vice versa [61]. The locality property is desirable in many types of applications as it improves their performance. In traditional database, preserving locality reduces the number of disk blocks to be fetched and seek time during query processing. Similarly, in DHT, preserving locality reduces the number of nodes visited when a query

is processed. Gotsman et. al. [61] and Jagadish [74] reported that in general, for any $d' < d$ it is not possible to always map two points that are close in a d-dimensional space to two points that are close in a d'-dimensional space[1]. Referring to Figure 4.2, two adjacent coordinates, (1, 0) and (2, 0), are mapped onto two identifiers that are further apart: 2 and 8 in z-curve, and 1 and 14 in Hilbert curve, respectively.

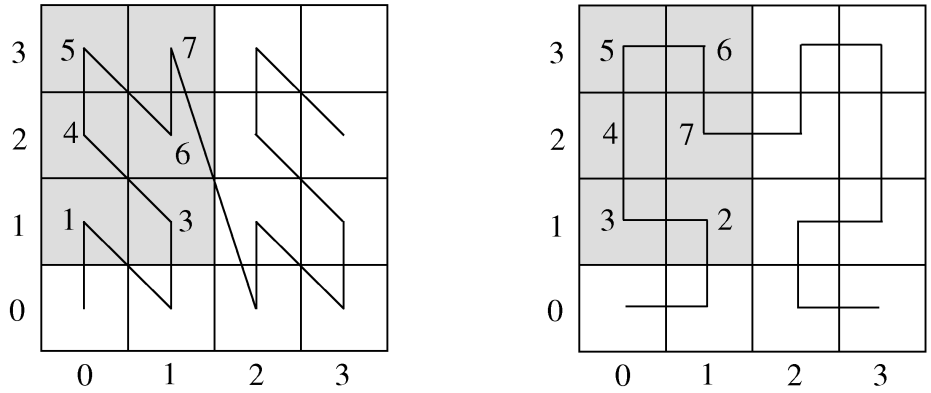
Though achieving optimal locality is not possible, studies have indicated that among various SFC, Hilbert SFC achieves better locality when applied to multi-dimensional indexing [74, 103]. Intuitively, this is because two consecutive Hilbert identifiers always connect two adjacent coordinate points. Jagadish [74] and Moon et. al. [103] quantify the locality-preserving property using the number of *clusters*. A cluster is a group of coordinate points, inside a d-dimensional *region* that are mapped onto consecutive identifiers. The region represents a multi-dimensional range query and is a subspace of a d-dimensional space. Using theoretical analysis and simulation, Hilbert curve is shown to minimize the average number of clusters compared to other types of SFC [74, 103].

Figure 4.3 shows a region, i.e. the shaded area, which is mapped by z-curve and Hilbert curve. With z-curve, the region is covered by two clusters: the first cluster consists of identifier 1 and the second cluster consists of identifers 3–7. With Hilbert curve, the region is covered by one cluster only, which consists of identifiers 2–7.

4.2.2 Constructing Hilbert Curve

To construct a Hilbert curve, we recursively divide a d-dimensional space until L *approximation levels*. At approximation level l, where $1 \leq l \leq L$, we divide the

[1] In the case of SFC, $d' = 1$.



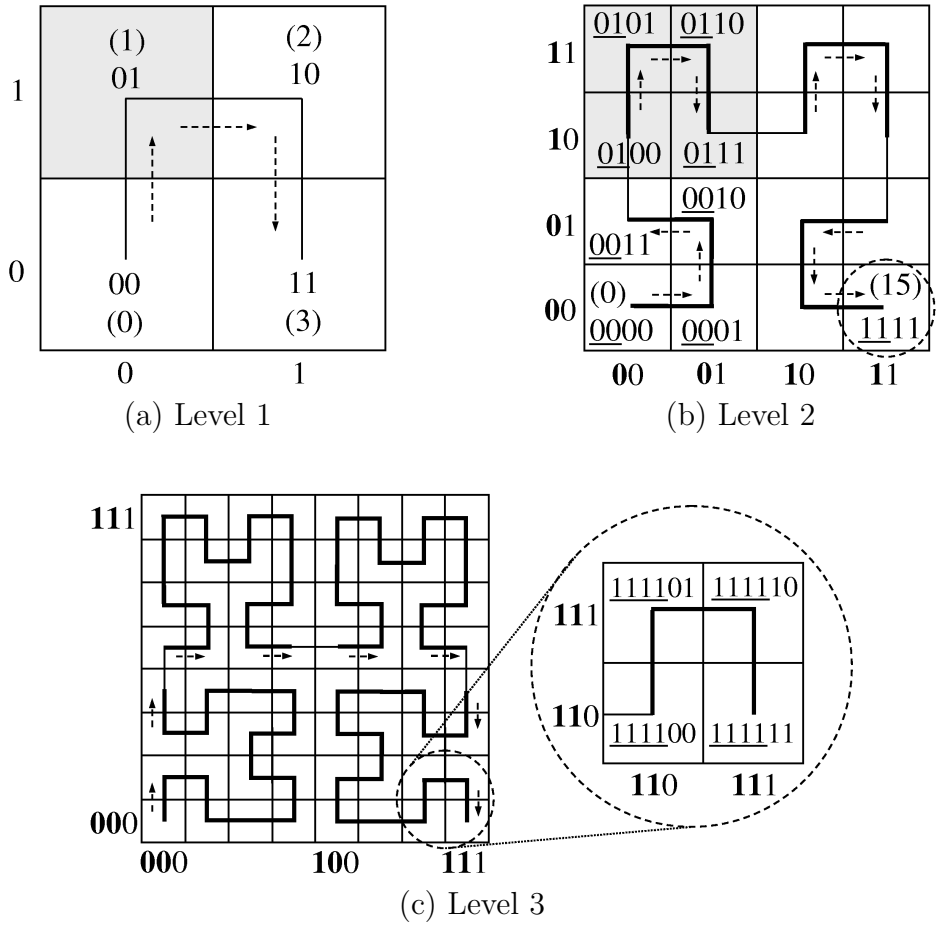
(a) z-Curve: Region is Covered by Cluster 1 and Cluster 3–7

(b) Hilbert Curve: Region is Covered by Cluster 2–7

Figure 4.3: Clusters and Region

d-dimensional space into 2^{dl} cells. The l^{th}-level SFC, which traverses the 2^{dl} cells, is constructed from a number of first-level curves, each of which being orientated differently. Figure 4.4 shows an example of constructing Hilbert SFC that covers a 2-dimensional space, up to approximation level 3. In this example, coordinate of cells and Hilbert identifiers are shown in binaries. For Hilbert identifiers, the decimal values are also shown in parentheses.

Figure 4.4a shows the level-1 curve which starts from coordinate $(0_2, 1_2)$, i.e. Hilbert identifier 0_2 (0), and ends at coordinate $(1_2, 1_2)$, i.e. Hilbert identifier 01_2 (3). In Figure 4.4b, each level-1 cell is split into four cells and a level-1 Hilbert curve, with a potentially different orientation, is applied on the four cells. For example, the four lower-left cells are covered by a level-1 Hilbert curve that has been rotated 270 degree along x-axis and mirrored along y-axis, whereas the four lower-right cells are covered by a level-1 Hilbert curve that has been rotated 90 degree along x-axis. To construct the next level of Hilbert curve, (Figure 4.4c), we follow the same process and then reuse level-2 curves. Thus, the eight lower-left cells, for example, are covered by a level-2 Hilbert curve that has been rotated 270 degree along x-axis and mirrored along y-axis.



(a) Level 1

(b) Level 2

(c) Level 3

Figure 4.4: Constructing Hilbert Curve on 2-Dimensional Space

The recursive construction of Hilbert curve results in the following three properties.

Property 4.1. *A cell at level $l-1$ is refined into 2^d subcells at level l.*

Proof. The cell is a region where the length of its dimensions are equal to one. The cell is refined to the next level by splitting each of its dimensions into two halves. This results in 2^d subcells in total. □

Based on Property 4.1, assuming that $1 \leq l \leq l' \leq L$, a cell at level l is equivalent to a region at level l' region, i.e. a group of level-l' cells. For example, the level-1 coordinate $(0_2, 1_2)$ in Figure 4.4a, is equivalent to level-2 coordinates $(00_2, 10_2)$, $(00_2, 11_2)$, $(01_2, 11_2)$, and $(01_2, 10_2)$ in Figure 4.4b.

Property 4.2. *Dimension values at approximation level l are l-bit long, where $l > 0$.*

Proof. We prove this property by induction.

If $l = 1$, each dimension consists of two values. Thus, one bit is needed to encode the values, and this proposition is true.

Assume that P_{l-1} is true for $1 < l \leq L$, i.e. $l - 1$ bits are needed to encode dimension values at level $l - 1$. We split each dimension value at level $l - 1$ into two subvalues at level l. Each level-l value is prefixed by the $(l - 1)$-bit value of its parent, and has one additional bit (0 or 1). Thus, this property is true. □

In Figure 4.4, the prefix of dimension values is shown in bold. Consider an example where the first-level cell $(0_2, 1_2)$, i.e. the shaded area in Figure 4.4a, is divided into four second-level subcells, i.e. the shaded area in Figure 4.4b. At the second-level cells, the possible values for x-axis are derived by concatenating the x-value of the parent cell with 0 and 1. The similar process applies for y-axis as well.

Property 4.3. *Hilbert identifiers at approximation level l are dl-bit long.*

Proof. Since there are 2^{dl} level-l coordinates to map, dl bits are required to encode all the coordinates.

When a cell at level $l - 1$ is refined into 2^d cells at the next level, the subcurve that covers the parent cell are also refined into a 2^d contiguous subcurve at level l. The resulted identifiers at level l are prefixed by their parent's Hilbert identifier.□

In Figure 4.4, the prefix of Hilbert identifiers is underlined. In the example, when the first-level cell $(0_2, 1_2)$ with Hilbert identifier 01_2 (Figure 4.4a) is refined into

four second-level cells, the resulted second-level Hilbert identifiers are 0100_2, 0101_2, 0110_2, and 0111_2; all of which are prefixed by the parent cell's Hilbert identifier, 01_2 (Figure 4.4b).

4.3 Design

As illustrated in Figure 4.5, Midas is divided into two main parts, namely *indexing scheme* and *query engine*. Each d-attribute resource is indexed as a key which is a Hilbert identifier. The key is further mapped onto an R-DHT node. A multi-attribute range query is first transformed into a number of exact queries using Hilbert SFC. These exact queries are further processed by the query engine to minimize the number of R-DHT lookups required.

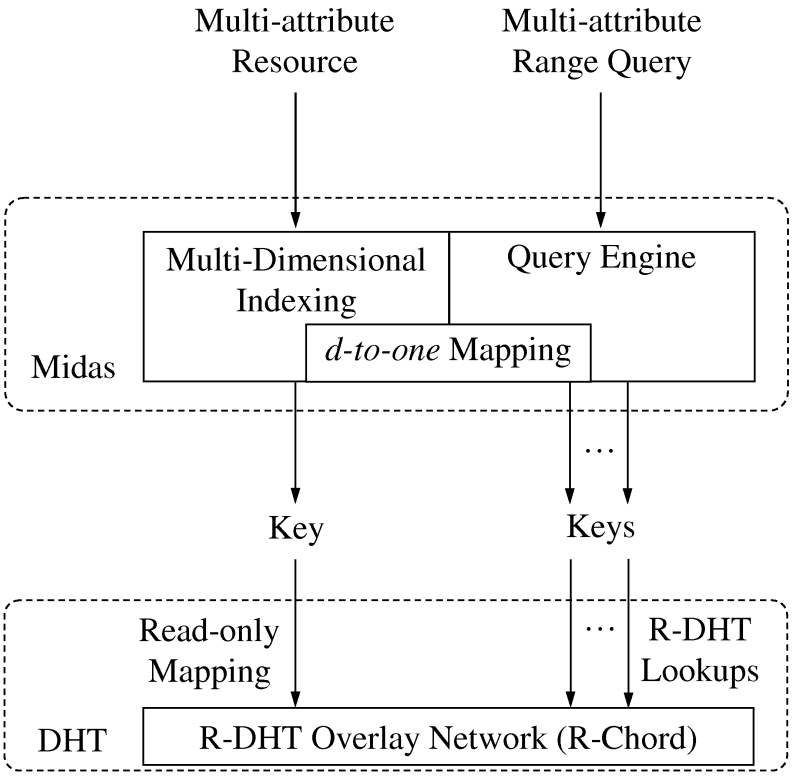


Figure 4.5: Midas Indexing and Query Processing

4.3.1 Multi-Attribute Indexing

Midas indexing consists of three basic components as shown in Figure 4.6. Firstly, it extracts the *type* of a resource, i.e. attributes of the resource (Definition 2.1), using two supporting components: *resource type specification* and *attribute-value normalization*. The resource type specification defines d attributes that constitute a resource type, e.g. attribute `cpu` and attribute `memory`. Then, the attribute-value normalization converts domain-specific attribute values into numbers, e.g. (`cpu`='*P4*', `memory`='*1 GB*') is normalized into resource type (`cpu`=2, `memory`=1). Once the type of a resource is derived, the d-to-*one* mapping maps the resource type onto a key using Hilbert SFC. Subsequently, the key is mapped onto an R-DHT node.

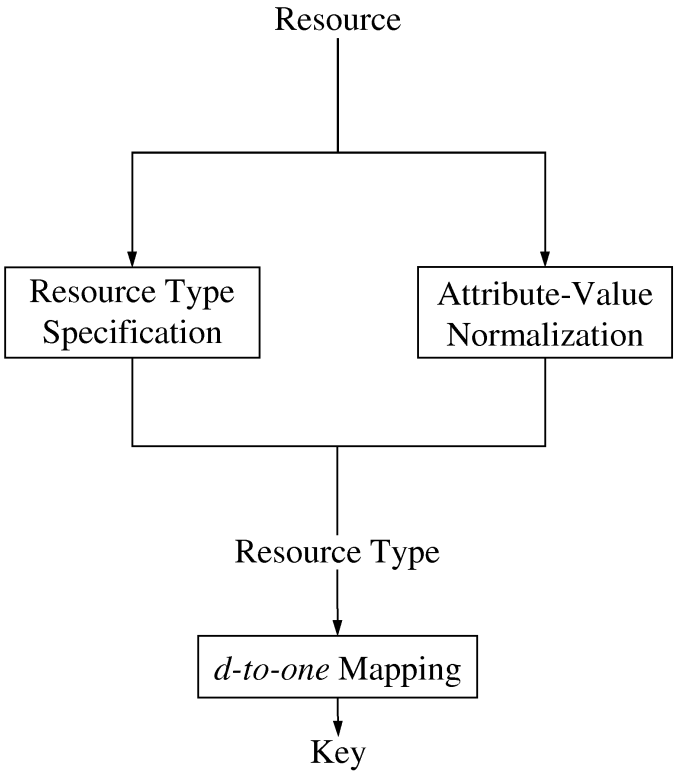


Figure 4.6: Midas Multi-dimensional Indexing

In the rest of this section, we first describe the d-to-*one* mapping, followed by the

two supporting components.

4.3.1.1 *d*-to-*one* Mapping Scheme

Based on its *type*, each resource is assigned a key which is a Hilbert identifier at the maximum approximation level. All resources are assumed to have the same number of attributes. These attributes can be derived based on a well-defined resource naming scheme such as GLUE schema [5] for compute resources (see Section 4.3.1.2 for more details).

Definition 4.1. *Let d denote the number of dimensions and m denotes the bit length of one-dimensional identifier space. A key in the identifier space is defined as an m-bit Hilbert code at the maximum approximation level L where $L = m/d$.*

Each resource is modeled as a point in a d-dimensional attribute space. The coordinate is determined by the resource type which consisting of d attributes. Each dimension represents an attribute encoded as an (m/d)-bit value, and thus, there are $2^{m/d}$ possible values per dimension (Figure 4.7). The m-bit Hilbert identifier of the coordinate will become the key assigned to the resource. Because the coordinate of a resource is determined by the resource type, resources of the same type occupies the same coordinate point and are assigned the same key (Definition 2.1). Finally, the key is mapped onto an R-DHT node using our read-only mapping scheme (Section 2.3.1–2.3.2).

The following example illustrates the process of indexing resources characterized by two attributes, namely `cpu` and `memory`, assuming m = 4-bit.

- *Assign Key to Resource*

  As illustrated in Figure 4.8a, resource r with `cpu` = *P4* and `memory` = *1 GB* is modeled as coordinate point (2, 1) in a 2-dimensional attribute space.

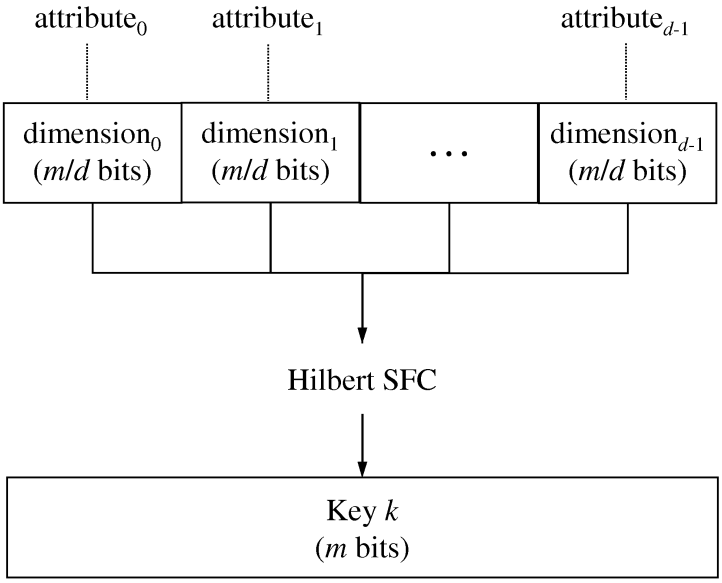


Figure 4.7: Attributes and Key

The coordinate is derived by normalizing[2] the two attributes into a resource type consisting of two attributes, namely `cpu` = 2 and `memory` = 1.

Using Hilbert SFC, coordinate (2, 1) is converted to Hilbert identifier 13 which is 4-bit long. Thus, r is assigned key $k = 13$.

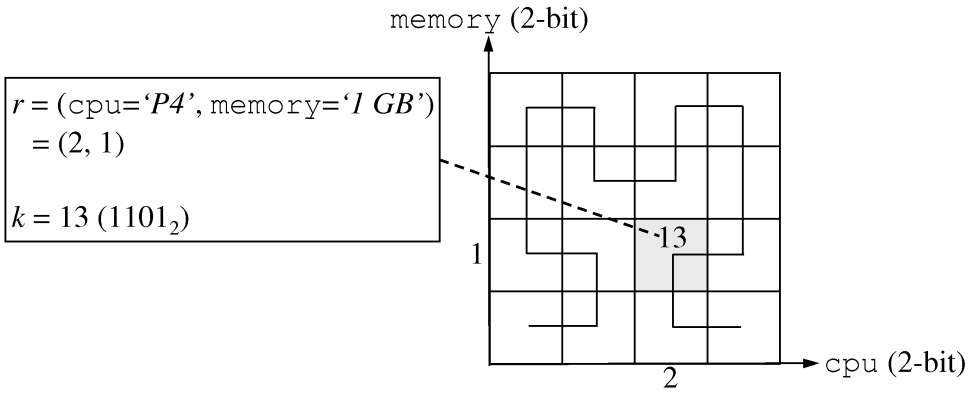
- *Map Key onto R-DHT Node Identifier*

  Assume that r is shared by host h. According to Definition 2.2, key 13 $\in T_h$ and the key is associated with node $n = 13|h$ (Figure 4.8b). If key 13 represents a new resource type of host h, then node joins an R-Chord overlay and occupies segment S_{13} (Figure 2.8 and Theorem 2.7). Otherwise, no new node is created on the overlay (Corollary 2.1).

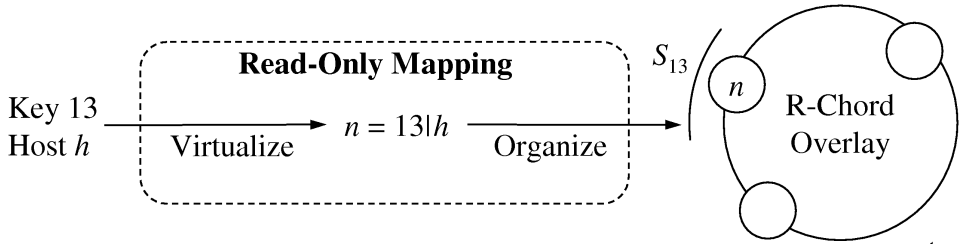
4.3.1.2 Resource Type Specification

The resource type specification defines d *indexing attributes*, e.g. indexed columns in traditional database, that constitute a resource type out of d' resource attributes($d \leq d'$). There are several reasons to index resource only by a subset of resource at-

[2]See Section 4.3.1.3 for the details.



(a) Assign Key 13 to Resource r



(b) Map SFC Key 13 onto R-DHT Node 13|h

Figure 4.8: Example of Midas Indexing ($d = 2$ Dimensions and $m = 4$ Bits)

tributes. Firstly, keys are kept stable by excluding attributes that change frequently. Otherwise, a resource must be re-assigned a new key when some of its attributes change. Secondly, we need to ensure that an (m/d)-bit dimension is sufficient to represent all possible values of an attribute. Thirdly, it has been shown that for higher dimension, the locality property of Hilbert SFC decreases, i.e. a higher number of clusters per query region [74, 103].

To include as many resource attributes as possible without significantly increasing the dimensionality d, we can combine several resource attributes into one *compound attribute* [86]. Consider a compound attribute (`attr`) that consists of i *member attributes*. A value of this attribute, which corresponds to one of $2^{m/d}$ dimension values, is denoted as a tuple of $\langle \texttt{member}_0, ..., \texttt{member}_{i-1} \rangle$. To support range queries, we impose an ordering on `attr` tuples where $\texttt{member}_j$ must be logically contained by $\texttt{member}_k$ $(j > k)$.

As an example, consider a compound attribute `book` with three member attributes called `chapter`, `section`, and `subsection`. We define each value of this compound attribute as a tuple of ⟨`chapter`, `section`, `subsection`⟩ since each subsection is part of a section, and each section is part of a chapter. Assuming member attributes are encoded as 2-bit values, each `book` tuple is encoded to a 6-bit dimension value derived by concatenating the three member attributes. Figure 4.9 shows the first chapter, i.e. all tuples with `chapter` = 00_2, is encoded to dimension values prefixed by 00_2 (i.e. dimension values 0–15). Similarly, the last section of the last chapter, i.e. all tuples with `chapter` = 11_2 and `section` = 11_2, are encoded to dimension values whose prefix is 1111_2 (i.e. dimension values 60–63).

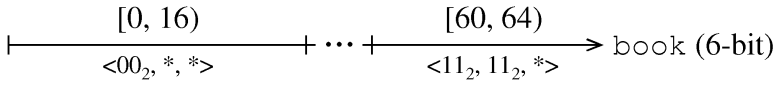


Figure 4.9: Dimension Values for Compound Attribute `book`

Table 4.2 shows an example of resource type specification based on GLUE schema [5] (Figure 4.10) to describe resource in a computational grid. We define a resource type using five attributes, out of more than 20 as specified in GLUE schema. Each of the attributes are modeled as a dimension with 32-bit long values. Thus, each key (Hilbert identifier) is 160-bit, which is a typical value used in Chord and several other DHT implementations. The specification includes two compound attributes, namely `OS` and `CPU`, each of which consists of three and four member attributes, respectively.

4.3.1.3 Normalization of Attribute Values

Each domain-specific attribute value is encoded as an (m/d)-bit length number (i.e. dimension value). For example, attribute `cpu` may consist of the following values: *P3*, *P4*, or *SPARC*. Each of these values is encoded as a number in the range of 0 to $2^{m/d}$. To support range queries, the normalization encodes attribute

```
<Cluster UniqueID="ce01.cr.cnaf.infn.it">
    <SubCluster UniqueID="ce01-lcg.cr.cnaf.infn.it">
        <Name>ce01-lcg.cr.cnaf.infn.it</Name>
        <PhysicalCPUs>0</PhysicalCPUs>
        <LogicalCPUs>0</LogicalCPUs>
        <TmpDir>/tmp</TmpDir>
        <WNTmpDir>/tmp</WNTmpDir>
        <OperatingSystem Name="SLC" Release="3.0.5" Version="3" />
        <Processor Model="DUAL-XEON" ClockSpeed="3000" Vendor="INTEL"
                   OtherDescription="String" InstructionSet="String" />
        <NetworkAdapter OutboundIP="true" InboundIP="false"/>
        <MainMemory VirtualSize="3072" RAMSize="2048"/>
        <Architecture SMPSize="2" PlatformType="String"/>
        <Benchmark SF00="400" SI00="1060"/>
        <Location LocalID="String">
                <Name>String</Name>
                <Version>String</Version>
                <Path>String</Path>
        </Location>
        <RunTimeEnv>
                <Variable>LCG-2_6_0</Variable>
                <Variable>VO-atlas-release-10.0.1</Variable>
                <Variable>VO-cms-ORCA_8_7_5</Variable>
        </RunTimeEnv>
    </SubCluster>
</Cluster>
```

Figure 4.10: Sample XML Document of GLUE Schema

values i and j to dimension values $f(i)$ and $f(j)$ such that $f(i) < f(j)$ if and only if $i < j$. We outline three approaches for the normalization of domain-specific attribute values.

1. *Static Conversion*

   This approach converts each domain-specific attribute value to a predefined dimension value, e.g. `cpu` = *P4* is converted to dimension value 2. The concept of static conversion has been applied in other fields as well. For example, LINUX operating system allocates a predefined number and name to each device, e.g. the first SCSI devices is allocated device number and device name 0 and `/dev/sda`, respectively.

   We can further extend the static conversion by mapping a group of attribute values (e.g. all `cpu` from a particular vendor) to contiguous dimension values. The administrative authority responsible for the attribute values (e.g. the manufacturer) manages the allocated range. This is analogous to the allocation of IP addresses in networking.

Dimension	**Length (bit)**	**Description**
Machine Count	32	Number of instances of the resource type
CPU Count	32	Number of processors per machine
Memory	32	Size of physical memory (multiplied by 256 MB)
OS	32	Operating system installed on a machine
Name	12	
Release	10	
Version	10	
CPU	32	Processor type of a machine
Vendor	7	
Architecture	5	
Model	5	
Clock Speed	15	CPU speed (multipled by 256 MHz)

Table 4.2: Resource Type Specification for Compute Resources based on GLUE Schema

2. *Locality-Preserving Hashing*

   A locality-preserving hash function [19, 28] is applied to each attribute value to obtain the corresponding dimension value. With locality-preserving hashing, similar attribute values are hashed onto dimension values that are also similar. Locality-preserving hashing supports range queries since $i < j < k$ is hashed to dimension values that satisfy condition $hash(i) < hash(j) < hash(k)$. If we use non-locality-preserving hashing, the condition $hash(i) < hash(j) < hash(k)$ is not guaranteed.

3. *Interval Mapping*

   For numerical attributes, attribute values can simply be divided into intervals and each interval i is directly mapped onto a dimension value. Thus, each dimension value represents an attribute value in a multiplicity of i.

4.3.2 Query Engine and Optimizations

A multi-attribute range query is transformed into search keys. A naive scheme treats each search key as an exact query (i.e. a DHT lookup). This results in many nodes visited during query processing. In Midas, we propose to minimize the number of nodes visited by initiating lookups only for search keys that represent available resources.

A multi-attribute range query specifies d search attributes where each of the attributes can be constrained by a range. A range imposes a limit on a search attribute using relational operators such as $<$, $\leq$, $=$, $>$, or $\leq$. After normalizing search attributes into dimension values, a query becomes a *region* in the d-dimensional attribute space (Definition 4.2). For $d = 2$, a query region resembles a rectangle and the number of search keys is equal to the area of the rectangle. For $d = 3$ and $d > 3$, a query region resembles a cube and a hypercube, respectively; the number of search keys is equal to the volume of the cube and the hypercube.

Definition 4.2. *A query region (Q) is represented with the two endpoints of its diagonal, namely $Q.lo$ and $Q.hi$. Endpoint $Q.lo$ refers to the smallest coordinate in the query region, i.e. the coordinate where each dimension consists of the smallest value in the range specified for the dimension. Similarly, endpoint $Q.hi$ refers to the largest coordinate in the query region.*

Figure 4.11 shows an an example of a 2-attribute range query: *find compute resources with* P3 $\leq$ ***cpu*** $\leq$ P4 *and* 1 GB $\leq$ ***memory*** $\leq$ 2 GB. The two ranges specify by the query are 1–2 and 0–1 for dimension `cpu` and dimension `memory`, respectively. The query region Q is illustrated as a shaded rectangle, where $Q.lo = (\texttt{cpu}_{min}, \texttt{memory}_{min}) = (1, 0)$ and $Q.hi = (\texttt{cpu}_{max}, \texttt{memory}_{max}) = (2, 1)$. Both $Q.lo$ and $Q.hi$ are not necessarily converted to the smallest and largest

Hilbert identifiers within the query region.

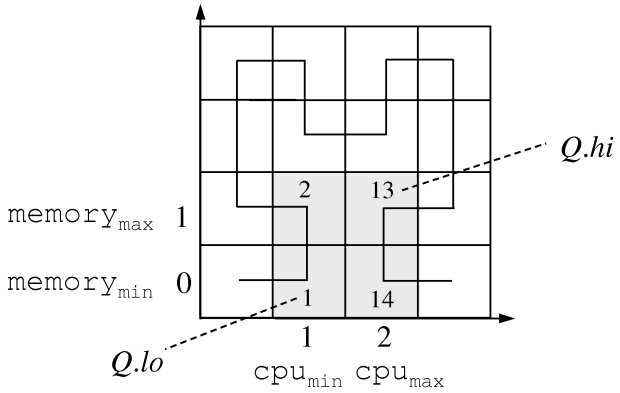


Figure 4.11: Range Query with Search Attributes `cpu` and `memory`

Each query is transformed into search keys, i.e. Hilbert identifiers assigned to all coordinates in the query region. For example, the query region covered by the shaded rectangle is transformed into four search keys grouped in two clusters, namely cluster 1–2 and cluster 13–14. Each search key is considered as an exact query. In a naive query processing, a query initiator (i.e. the user) issues one lookup per search key. However, this is not efficient for the following reasons:

1. The naive search includes unnecessary lookups for search keys that do not represent resources. Figure 4.12 shows two unnecessary lookups for search keys 13 and 14, out of four lookups. Because search keys 13 and 14 do not correspond to resources, there are no S_{13} and S_{14} in the underlying R-Chord overlay. As a result, both the unnecessary lookups terminate at a different segment, S_{15}.

2. The naive search ignores the clustering property of Hilbert SFC. Since search key 1 and 2 are clustered, the closer proximity between S_1 and S_2 can be exploited by issuing only *lookup(1)* and letting S_1 forward a request to S_2.

To support efficient query processing, we propose an incremental search strategy

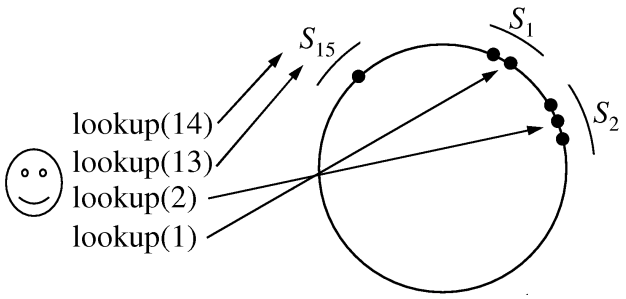


Figure 4.12: Naive Search Algorithm

that processes only *available keys*. A key is available if the resource type represented by the key consists of at least one resource instance. Figure 4.13 shows Midas incremental search. After obtaining q, i.e. an ordered set of search keys (line 2 in Figure 4.13a), the algorithm initiates *lookup*(k) for the lowest key $k \in q$ (line 4). This lookup will end-up at node n in segment $S_{k'}$ (line 5), where $S_{k'}$ is the succeeding segment of k. If $k' = k$ then we add k (or the associated key-value pair) to the result set, otherwise we discard k (line 3–9 in Figure 4.13b). Prior to continuing the incremental search, we remove any search key k" that does not correspond to any resources (line 10–11).

A search key is eliminated subject to one of the following conditions:

1. $preceding\_segment(k) < k" < k'$

   We eliminate any unavailable key k" that precedes k', i.e. key k" which is within the left-side range of node n as shown in Figure 4.14. The reason is that in R-DHT, $S_{k'}$ is the succeeding segment of k" (and k) only if k" does not exist, otherwise the succeeding segment would be $S_{k"}$.

   To quickly obtain the preceding segment, R-Chord *lookup*(k) can be modified to return not only the succeeding segment of k, but also the preceding segment of k.

2. $k' < k" < succeeding\_segment(k')$

   We eliminate any unavailable key k" that succeeds k', i.e. k" is within the

```
1.  h.mdq_search(Query Q)
2.      q = transform_query_region(Q.lo, Q.hi);
3.      rs = {};
4.      k = get_min(q);
5.      (n, p) = lookup(k);
6.      return n.incr_search(q, rs, p);
```

(a) Main Algorithm

```
1.  n.incr_search(List q, ResultSet rs, PreceedingSegment p)
2.      //Check if h owns a key equals to the search key
3.      k = get_min(q);
4.      T_h = a set of keys in h;
5.      for each y ∈ T_h do
6.          if y == k then
7.              rs = rs ∪ {(k, n)};
8.
9.      q = q − {k};
10.     q = eliminate_keys(q, p, prefix(n));
11.     q = eliminate_keys(q, prefix(n), prefix(succ_seg));
12.
13.     if q == {} then
14.         return rs;
15.
16.     //Search the next lowest key
17.     k = get_min(q);
18.     (n', p) = lookup(k);
19.     return n'.incr_search(q, rs, p);
20. // Eliminate keys in the range of [low, high)
21. n.eliminate_keys(List q, Key low, Key high)
22.     k = get_min(q);
23.     while k ≠ nil and low < k < high do
24.         q = q − {k};
25.         k = get_min(q);
26.
27.     return q;
```

(b) Helper Functions

Figure 4.13: Midas Incremental Search Algorithm

right-side range of node n in Figure 4.14.

To quickly locate $S_{k'}$, each node maintains a pointer to its succeeding segment. This new pointer can be considered as a finger and is put in the

finger table. The pointer is maintained through periodic stabilization. With this new pointer, locating the succeeding segment of a node can be done by simply examining the node's finger table instead of issuing *lookup*$((S_k{+}1)|0)$.

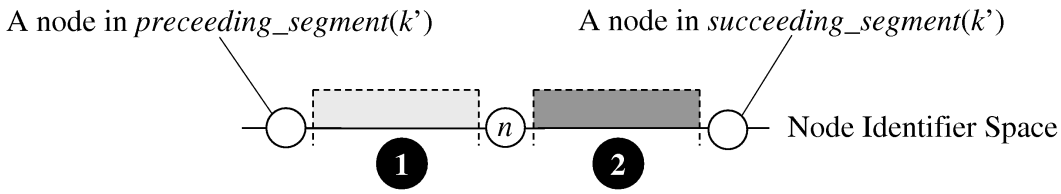


Figure 4.14: Search-Key Elimination

Figure 4.15 illustrates an example of incremental search for the query illustrated in Figure 4.11. Given the four search keys, 1, 2, 13, and 14, Midas initiates a DHT lookup for the lowest key 1. Since the lookup finds the key at segment S_1, Midas adds key 1 to the result set and continues with *lookup*(*2*). As this lookup arrives at S_2, Midas adds key 2 to the result set. Furthermore, it eliminates key 13 and 14 since the succeeding segment of S_2 is S_{15}. Thus, the final result set consists of two keys: 1 and 2. To return faster results, query processing can be parallelized by partitioning the search keys and performing one incremental search per partition.

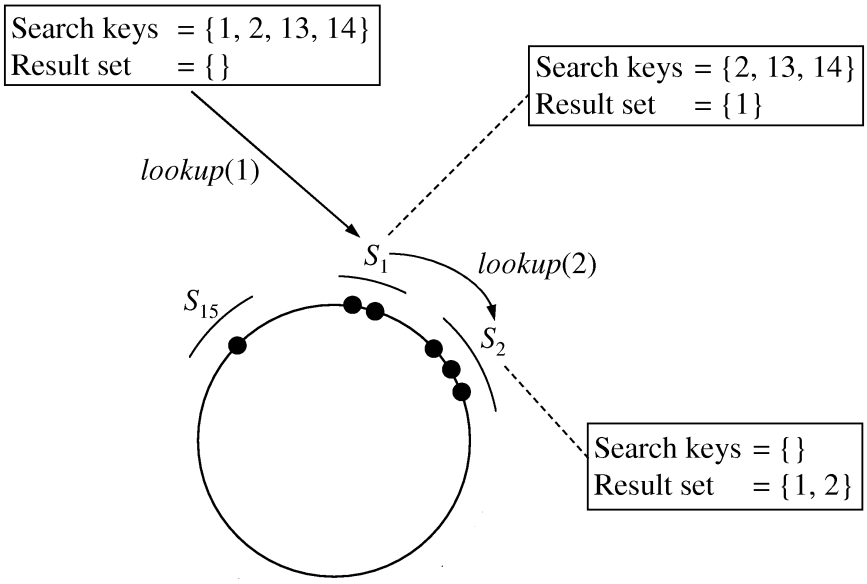


Figure 4.15: Example of Range Query Processing

4.4 Performance Evaluation

Using simulation, we evaluate the performance of multi-attribute range queries on DHT through simulations[3]. We first show the efficiency of Midas compared to the naive query processing scheme, and the impact of underlying overlay (R-DHT and conventional DHT) to the performance of Midas. Next, we study the impact of data-item distribution on the performance of multi-attribute range queries. We compare R-Chord-based Midas and Chord-based Midas and measure query cost, query resiliency to node failures, and query performance under churn.

The implementation of Midas on Chord and R-Chord differs in the search-key elimination algorithm:

1. Consider a *lookup*(k) request that ends up at node n (i.e. the successor of k). Node n eliminates only search key k" where $n.predecessor < k" < n$; this is similar to the first condition described in Section 4.3.2. However, n does not eliminate the search key if $n < k" < n.successor$, which is equivalent to the second condition in R-Chord, because Chord maps k" to $n.successor$. Since $n.successor$ is the responsible node of k", it is the one who is responsible to eliminate the key.

2. When eliminating key k", node n also checks if key k" is actually available and stored on it. If k" is available, it is added to the result set.

Unless stated otherwise, our experiments use the following parameters:

- d, the number of attributes per resource, is varied from 3 to 5. Each dimension is 6-bit long (m/d), and thus, is capable to hold $2^6 = 64$ dimension

[3]Our simulator uses Lawder's table-driven algorithm to perform the Hilbert mapping [84]. The algorithm is applicable to arbitrary number of dimensions, in contrast to several earlier algorithms [26, 36, 92] which are limited to 2-dimensional space. Recently, Jin et. al. [75] proposed a table-driven framework capable of constructing different types of SFC.

values.

- m, the number of bits for keys and host identifiers, is $6d$-bit.
- K, the number of unique keys (i.e. resource types), is varied from 5,000 to 150,000. This means, there are K points in the d-dimensional attribute space. We generate these points using the normal distribution.
- N, the number of hosts, is varied from 25,000 to 50,000. Each host shares $T = 8$ unique resource types on average (i.e. $|T_h| \sim U[4, 12]$ unique resource types), and each resource type may be offered for sharing by more than one hosts.
- The size of a range query, i.e. the number of search keys, is a^d where a ($\leq 2^{m/d}$) is a length parameter. Queries are classified based on their shape:
  - Type 1, i.e. $(a)^d$, are query regions where the length of each dimension is a.
  - Type 2, i.e. $(0.5a)(2a)(a)^{d-2}$, are query regions where the length of the first dimension and the second dimension are $0.5a$ and $2a$, respectively, while the length of the remaining dimension is a.

4.4.1 Efficiency

To study the improvement by Midas over the naive scheme, we compare the average number of nodes visited per query, i.e. responsible nodes, which store available keys, and intermediate nodes. The naive scheme initiates one lookup per search key, whereas Midas initiates lookups only for available keys. Both schemes use R-Chord as the underlying overlay in a system comprising 25,000 hosts (N) . For each value of length parameter a, we simulate 1,000 queries consisting of 500 type-1 queries and 500 type-2 queries. Each query consists of Q_{skey} ($= a^d$) search keys

and Q_{akey} ($\leq Q_{skey}$) available keys. Table 4.3 presents the query profile and the simulation results.

K	a	$d = 3$		$d = 4$		$d = 5$	
		Q_{skey}	Q_{akey}	Q_{skey}	Q_{akey}	Q_{skey}	Q_{akey}
5,000	4	64	7	256	1	1,024	0.1
	8	512	56	4,096	13	32,768	3
	16	4,096	442	65,536	206	1,048,576	97
50,000	4	64	46	256	8	1,024	1
	8	512	375	4,096	125	32,768	29
	16	4,096	3,058	65,536	2,068	1,048,576	379

(a) Query Profile

K	a	$d = 3$		$d = 4$		$d = 5$	
		Naive	**Midas**	**Naive**	**Midas**	**Naive**	**Midas**
5,000	4	337	27	1,391	21	5,654	18
	8	2,697	115	22,381	85	182,151	73
	16	21,672	627	360,068	533	5,870,126	499
50,000	4	403	75	1,828	53	7,365	39
	8	3,243	474	29,440	344	238,941	245
	16	25,922	3,386	471,964	3,267	6,431,520	2,670

(b) Average Number of Nodes Visited per Query

Table 4.3: Performance of Query Processing in Naive Scheme vs Midas

The result in Table 4.3b shows that Midas is more efficient than the naive scheme in processing multi-attribute range queries, because Midas initiates lookups only for available keys. Our result reveals that the number of nodes visited in Midas is at least five times (i.e. $d = 3$, $K = 50,000$, and $a = 4$) smaller than the naive scheme. In the naive scheme, the cost is determined by the size of the query region, i.e. $\Omega(Q_{skey})$. Because one R-Chord lookup is initiated per search key and each lookup visits $O(\min(\log K, \log N))$ nodes according to Theorem 2.2, the number of nodes visited for each query is $O(Q_{skey} \min(\log K, \log N))$. On the other hand, the cost of query processing in Midas is determined by the number of available keys

(Q_{akey}) which is less Q_{skey}. Because Midas looks up only for the available keys, the number of nodes visited is at least Q_{akey}. In addition, due to incremental search, the cost of the initial lookup (for the smallest search key) becomes insignificant as Q_{akey} increases. Thus, the number of nodes visited per query is $\Omega(Q_{akey})$.

4.4.2 Cost of Query Processing

The impact of data-item distribution on Midas query processing cost is evaluated using both Chord and R-Chord. The query processing cost is measured using the average number of nodes visited per query. We simulate 10,000 type-1 queries and 10,000 type-2 queries on system with 25,000 to 50,000 hosts. The size of each query is kept constant at $Q_{skey} = 16^d$. Table 4.4 shows the query profile in terms of the number of available keys per query (Q_{akey}), and the simulation results.

Table 4.4b shows that the query cost in R-Chord-based Midas is affected by K, whereas Chord-based Midas is affected by N. Because each R-Chord node is responsible for its own key only, the number of nodes visited is $\Omega(Q_{akey})$. As Q_{akey} increases when K is increased, so does the number of nodes visited. In Chord, the number of nodes visited is $\Omega(Q_{cnode})$ where Q_{cnode} denotes the number of Chord nodes responsible for available keys. Due to data-item distribution, each Chord node is responsible for one or more keys, and thus, $Q_{cnode} \leq Q_{akey}$ (Figure 4.16). As N is increased, the value of Q_{cnode} increases because available keys are distributed to a higher number of Chord nodes. This is shown in Table 4.5 which compares Q_{cnode} in Chord rings consisting of $25,000$ nodes (N_{25}) and 50,000 nodes (N_{50}). A similar observation regarding the performance of range queries on conventional DHT has also been made by Cristina et. al. [128].

Though Table 4.4b shows that the query cost in R-Chord is higher than Chord as K is increased, it does not contradict our earlier analysis on R-Chord lookup

K	$d = 3$	$d = 4$	$d = 5$
5,000	445	211	95
7,500	668	325	150
10,000	879	432	199
15,000	1,271	643	299
25,000	1,938	1,042	475
50,000	3,057	2,066	953
75,000	3,416	2,897	1,359
100,000	3,388	3,572	1,669
125,000	3,206	4,070	1,929
150,000	2,992	4,510	2,137

(a) Average Number of Available Keys per Query ($Q_{skey} = 16^d$ Search Keys)

N	K	$d = 3$		$d = 4$		$d = 5$	
		Chord	R-Chord	Chord	R-Chord	Chord	R-Chord
25,000	5,000	650	629	418	537	358	499
	7,500	647	885	423	741	358	682
	10,000	648	1,120	420	928	360	836
	15,000	641	1,544	420	1,274	358	1,120
	25,000	649	2,242	416	1,887	363	1,589
	50,000	638	3,387	412	3,312	357	2,675
	75,000	667	3,741	414	4,385	364	3,513
	100,000	655	3,708	418	5,225	358	4,137
	125,000	644	3,526	424	5,839	357	4,631
	150,000	654	3,313	421	6,352	357	5,003
50,000	5,000	1,133	627	662	534	524	498
	7,500	1,128	889	661	740	520	681
	10,000	1,124	1,123	661	927	521	833
	15,000	1,127	1,545	660	1,270	519	1,116
	25,000	1,138	2,238	661	1,882	524	1,600
	50,000	1,129	3,447	662	3,360	522	2,733
	75,000	1,154	3,993	663	4,675	525	3,720
	100,000	1,148	4,194	664	5,833	520	4,624
	125,000	1,118	4,195	665	6,895	520	5,393
	150,000	1,134	4,121	665	7,783	519	6,079

(b) Average Number of Nodes Visited

Table 4.4: Query Cost of Midas

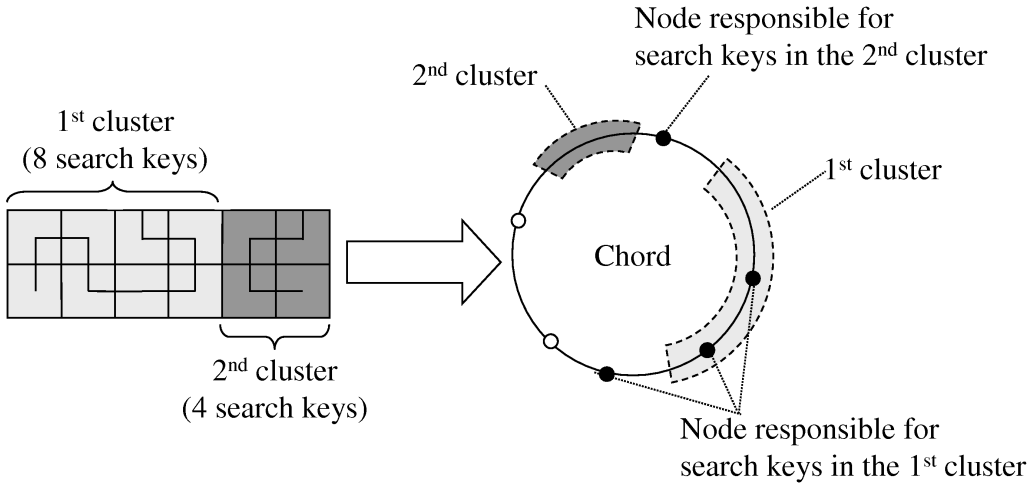


Figure 4.16: Four Chord Nodes are Responsible for Twelve Search Keys

K	$d = 3$		$d = 4$		$d = 5$	
	N_{25}	N_{50}	N_{25}	N_{50}	N_{25}	N_{50}
5,000	222	296	92	118	47	55
7,500	271	383	114	149	58	71
10,000	310	453	129	173	67	83
15,000	351	553	147	205	79	101
25,000	407	677	168	247	95	125
50,000	457	817	195	302	112	157
75,000	499	895	207	331	126	176
100,000	493	911	216	347	128	187
125,000	478	889	226	362	131	195
150,000	483	902	225	369	135	200

Table 4.5: Q_{cnode}

performance (Theorem 2.2) which states that the path length of each R-Chord lookup is at most equal to Chord. Instead, the higher cost of query processing in R-Chord is caused by a higher number of lookup operations (Table 4.6). As stated earlier, one lookup is required to locate each responsible node, and the number of responsible nodes in R-Chord (i.e. Q_{akey}) is higher than Chord (i.e. Q_{cnodes}). However, the number of intermediate hops per R-Chord lookup[4] is lower

[4]In both Chord and R-Chord, the number of intermediate hops per lookup decreases as K is increased, which is explained as follows. The query cost in Chord and R-Chord is $\Omega(Q_{cnode})$ and $\Omega(Q_{akey})$, respectively. Both Q_{cnode} and Q_{akey}, which also denote the number of nodes responsible for query results, increase as K is increased. Given a constant size of overlay network, a larger number of responsible nodes reduces the distance between responsible nodes.

N	K	$d = 3$		$d = 4$		$d = 5$	
		Chord	**R-Chord**	**Chord**	**R-Chord**	**Chord**	**R-Chord**
25,000	5,000	505	479	266	304	191	231
	7,500	357	704	209	441	153	326
	10,000	367	918	193	567	147	407
	15,000	362	1,300	202	807	149	561
	25,000	503	1,965	264	1,253	193	826
	50,000	363	2,958	194	2,256	147	1,424
	75,000	390	3,158	200	2,922	146	1,840
	100,000	509	3,396	265	3,926	187	2,418
	125,000	355	2,768	204	3,778	147	2,325
	150,000	347	2,516	196	4,040	148	2,497
50,000	5,000	918	477	446	303	296	230
	7,500	913	705	444	440	292	326
	10,000	909	917	444	566	293	406
	15,000	911	1,306	443	806	291	561
	25,000	921	1,962	444	1,249	295	832
	50,000	914	3,122	445	2,387	292	1,515
	75,000	936	3,648	446	3,444	293	2,134
	100,000	931	3,848	445	4,412	289	2,721
	125,000	904	3,857	447	5,307	290	3,232
	150,000	919	3,791	447	6,081	290	3,696

Table 4.6: Average Number of Lookups per Query (based on Table 4.4b)

than Chord for various d and K (Table 4.7). Overall, the results in Table 4.6–4.7 further emphasizes that the cost of query processing in R-Chord is higher due to the absence of data-item distribution instead of the cost of each (primitive) lookup operation.

The result in Table 4.7 also indicates sthat the clustering property of Hilbert SFC becomes poorer as dimensionality d is increased. On higher dimensions, locality preservation decreases where two resources that are semantically similar are assigned keys that are farther apart. These keys are further mapped onto responsible nodes whose distance is farther apart; this increasing the number of intermediate nodes per lookup. As a result, the path length of each lookup becomes longer and the overall query cost increases.

N	K	$d = 3$		$d = 4$		$d = 5$	
		Chord	**R-Chord**	**Chord**	**R-Chord**	**Chord**	**R-Chord**
25,000	5,000	0.85	0.38	1.23	1.07	1.62	1.75
	7,500	0.75	0.31	1.14	0.95	1.57	1.63
	10,000	0.67	0.26	1.09	0.88	1.53	1.56
	15,000	0.58	0.21	1.02	0.78	1.47	1.46
	25,000	0.48	0.15	0.94	0.67	1.39	1.35
	50,000	0.37	0.11	0.83	0.53	1.30	1.16
	75,000	0.32	0.09	0.79	0.46	1.24	1.07
	100,000	0.32	0.09	0.76	0.42	1.23	1.02
	125,000	0.33	0.10	0.73	0.40	1.21	0.98
	150,000	0.35	0.11	0.73	0.38	1.18	0.95
50,000	5,000	0.91	0.38	1.22	1.07	1.59	1.75
	7,500	0.82	0.31	1.15	0.94	1.54	1.63
	10,000	0.74	0.26	1.10	0.87	1.49	1.56
	15,000	0.63	0.21	1.03	0.78	1.44	1.46
	25,000	0.50	0.15	0.93	0.68	1.35	1.34
	50,000	0.34	0.11	0.81	0.53	1.25	1.16
	75,000	0.28	0.10	0.74	0.45	1.19	1.06
	100,000	0.25	0.09	0.71	0.41	1.15	1.00
	125,000	0.25	0.09	0.68	0.37	1.12	0.95
	150,000	0.25	0.09	0.66	0.35	1.10	0.91

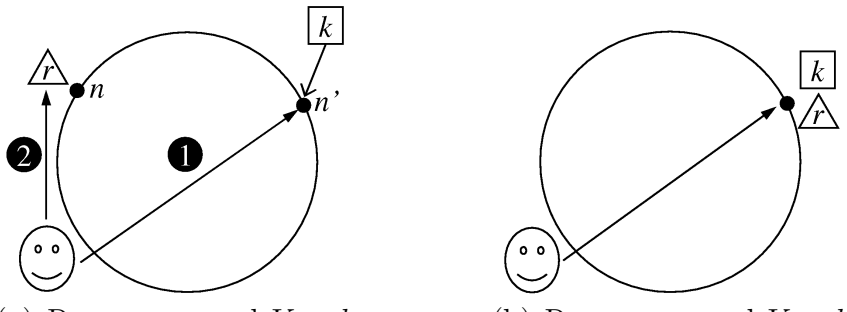
Table 4.7: Average Number of Intermediate Nodes per Lookup (based on Table 4.4b)

Our evaluation raises two implications to consider in reducing the cost of R-DHT query processing:

1. *Combine Query Processing and Resource Accesses*

   An R-DHT implementation supports this feature by providing an API that combines a lookup request and a resource-access request into a single request.

   The query cost presented in Table 4.4b excludes the additional hops needed by Chord to access resources once resource indexes have been found through query processing. As illustrated in Figure 4.17a, Chord maps key k belonging to node n onto another node n'. To access resource r that is assigned key k, a user first locates k stored at n' (step 1), before accessing r at node n (step 2). The total number of hops to access all resources that match a query is equal to $\Omega(Q_{akey})$. Therefore, the total cost in Chord-based Midas becomes the sum of the query cost in Table 4.4b and Q_{akey} (Table 4.4a). For example, when $N = 25,000$ hosts, $K = 5,000$ unique keys, and $d = 4$ dimensions, the total cost is $537 + 211 = 748$ hops. R-Chord, on the other hand, allows a user to access resources during query processing because resource r and its key k are located at the same node (Figure 4.17b).



(a) Resource r and Key k are Separated (Chord)

(b) Resource r and Key k are Located at the Same Node (R-Chord)

Figure 4.17: Locating Key and Accessing Resource in R-Chord and Chord

2. *Distributing Data Items to Trusted Nodes*

   We have identified that data-item distribution reduces the cost of query processing. Thus, introducing selective data-item distribution into R-DHT is a logical choice in optimizing its range query processing. R-DHT facilitates selective data-item distributions by grouping publicly-writable nodes in a reserved segment (Appendix B). The reserved segment is essentially another Chord ring embedded in a larger R-Chord ring. Query processing will search only among these trusted nodes. Assuming that the number of nodes in the reserved segment is denoted as Q_{cnode}, the cost to locate available keys in the reserved segment is $\Omega(Q_{cnode})$. Our experiment on Chord-based Midas is an extreme case of selective data-item distributions, where every node in the system is a trusted, writable node.

4.4.3 Resiliency to Node Failures

Query processing is resilient if it is able to locate available keys in the presence of node failures. To evaluate query resiliency, we simulate range queries when a percentage (F) of 25,000 hosts and 50,000 hosts fail simultaneously, and we measure the percentage of available keys that are successfully retrieved. The result shown in Table 4.8 shows that range-query resiliency is higher in R-Chord where nearly all available keys are retrieved.

The result of this experiment is consistent with our findings on the R-DHT lookup resiliency (Section 2.5.2). Because each R-Chord node is responsible only for its own keys, when a node fails, only its own keys are affected. And by its design (i.e. routing by segments and finger flexibility through backup fingers), R-Chord can locate a key as long as there is at least one alive node still sharing the key. On the other hand, Chord stores a key belonging to one node on another responsible node. When the responsible node fails, Chord fails to locate the keys (i.e. resource

F	K	$d = 3$		$d = 4$		$d = 5$	
		Chord	**R-Chord**	**Chord**	**R-Chord**	**Chord**	**R-Chord**
25%	5,000	74	>99	64	>99	75	>99
	7,500	68	>99	75	>99	74	>99
	10,000	72	>99	68	>99	67	>99
	15,000	70	>99	73	>99	72	>99
	25,000	70	>99	65	>99	74	>99
	50,000	76	>99	73	99	76	99
	75,000	80	99	77	97	65	97
	100,000	79	98	77	97	84	96
	125,000	79	97	81	96	83	95
	150,000	77	97	77	95	85	95
50%	5,000	28	96	30	95	33	95
	7,500	31	97	33	96	33	95
	10,000	28	97	31	96	35	96
	15,000	22	97	27	96	26	95
	25,000	34	98	33	97	36	96
	50,000	39	97	35	94	43	93
	75,000	36	93	39	89	32	88
	100,000	39	89	43	84	49	82
	125,000	47	86	52	81	42	78
	150,000	41	83	51	74	28	75

(a) $N = 25,000$ Hosts

Table 4.8: Percentage of Keys Retrieved under Simultaneous Node Failures

F	K	$d = 3$		$d = 4$		$d = 5$	
		Chord	**R-Chord**	**Chord**	**R-Chord**	**Chord**	**R-Chord**
25%	5,000	69	>99	72	>99	72	>99
	7,500	73	>99	70	>99	66	>99
	10,000	68	>99	65	>99	72	>99
	15,000	71	>99	70	>99	71	>99
	25,000	71	>99	70	>99	76	>99
	50,000	72	>99	72	>99	74	>99
	75,000	70	>99	69	>99	74	99
	100,000	72	>99	77	99	74	99
	125,000	73	99	75	98	72	98
	150,000	77	99	74	98	75	97
50%	5,000	24	97	31	95	31	95
	7,500	26	97	30	95	21	97
	10,000	25	97	25	95	31	95
	15,000	30	97	29	96	37	95
	25,000	25	98	30	96	36	95
	50,000	27	99	29	96	32	96
	75,000	30	99	32	96	41	95
	100,000	32	98	34	94	36	94
	125,000	29	96	37	91	37	91
	150,000	36	94	32	89	47	88

(b) $N = 50,000$ Hosts

Table 4.8: Percentage of Keys Retrieved under Simultaneous Node Failures

indexes) stored on the responsible node, even if the originating node of the keys (i.e. the node where the actual resources are located) is still alive.

4.4.4 Query Performance under Churn

To evaluate the performance range query processing under churn, we compare Midas on R-Chord and Chord overlays that change dynamically. The procedure is similar to the churn experiment in Section 2.5.4). We begin the experiments by warming up 25,000 hosts. In the next one-hour period, we simulate churn events (i.e. arrivals, fails, and leaves) produced by 25,000 hosts. Thus, there will be $N \sim 25,000$ alive hosts at any time within this duration. During this one-hour period, we also simulate a number of range query events, where the ratio of arrive:fail:leave:query is set at 2:1:1:1. Assuming that these events follow a Poisson distribution, we derive two rates to represent churn rate, $\lambda_B = 5$ events/second and $\lambda_G = 17$ events/second, based on the measurements on peer life-time by Bhagwan et. al. [25] and Gummadi et. al. [63], respectively (refer to Section 2.5.4 for details of the derivation). Each node in the overlay invokes the finger correction every 60 seconds on average. Table 4.9 presents the percentage of available keys that are successfully retrieved.

With the moderate churn rate (Table 4.9a), our result shows that R-Chord performs reasonably well compared with Chord. Though R-Chord overlay is eight times larger than Chord, the number of available keys retrieved in R-Chord is, at most, 10% lower than Chord. With the high churn rate (Table 4.9a), the number of keys retrieved in R-Chord is up to 25% lower than Chord. This result again shows that R-DHT lookup performance under churn is influenced by finger flexibility. When K is increased, finger flexibility is reduced, i.e. each segment in the overlay consists of a small number of nodes (see Theorem 2.6). As there are not enough stable nodes within each segment, the effectiveness of backup fingers is

K	$d = 3$		$d = 4$		$d = 5$	
	Chord	**R-Chord**	**Chord**	**R-Chord**	**Chord**	**R-Chord**
5,000	98	98	98	98	98	98
7,500	98	98	96	98	98	97
10,000	96	97	97	97	96	97
15,000	95	96	95	98	96	95
25,000	94	95	95	93	95	93
50,000	91	91	90	88	91	89
75,000	95	87	92	86	93	81
100,000	95	86	96	81	90	87
125,000	90	88	98	87	92	80
150,000	94	87	98	77	93	83

(a) $\lambda_B = 5$ Events/Second

K	$d = 3$		$d = 4$		$d = 5$	
	Chord	**R-Chord**	**Chord**	**R-Chord**	**Chord**	**R-Chord**
5,000	95	94	93	92	93	90
7,500	92	91	87	88	88	89
10,000	90	88	89	87	89	87
15,000	87	86	84	84	87	83
25,000	84	80	82	77	85	75
50,000	81	67	73	62	75	61
75,000	79	63	80	60	82	60
100,000	74	63	78	54	82	58
125,000	78	59	78	52	84	56
150,000	81	59	82	51	79	55

(b) $\lambda_G = 17$ Events/Second

Table 4.9: Percentage of Keys Retrieved under Churn ($N \sim 25,000$ Hosts)

reduced under churn because nodes have a higher number incorrect fingers (i.e. fingers which point to incorrect segments). This leads to the resiliency of R-DHT, which is due to segment-based overlay, is also reduced. However, in terms of the number of keys retrieved, R-Chord can still retrieve at least half of available keys.

In summary, the performance of Midas under churn is mainly influenced by the effectiveness of backup fingers. With a higher finger flexibility, i.e. a higher number of nodes per segment, backup fingers is effective in increasing the resiliency of

routing by segment. However, when each segment has a small number of nodes, our results implies that it is important for each node to maintain fingers pointing to stable nodes. One approach to address this is by using a hierarchical R-DHT where only stable nodes can become supernodes.

4.5 Summary

We have presented Midas, a scheme to support multi-attribute range queries on R-DHT. Using Hilbert SFC, Midas assigns to each d-attribute resource a one-dimensional key. In processing a range query, Midas performs search-key elimination to avoid issuing unnecessary lookups, and incremental search to exploit the clustering property of Hilbert SFC. Due to its read-only property, R-DHT does not need to send additional requests to access resources, as resources and its key are co-located.

Performance evaluation of Midas is conducted through simulations, and the main results are as follows.

Efficiency of Midas Compared to a naive scheme of query processing, Midas significantly reduces the number of nodes visited in processing a multi-attribute range query. To process a query consisting of Q_{skey} search keys and Q_{akey} ($\leq Q_{skey}$) answers, Midas visits $\Omega(Q_{akey})$ nodes whereas the naive scheme requires $\Omega(Q_{skey})$ nodes. We further validate the efficiency of Midas through simulations. Our experiments reveal that Midas is at least five times more efficient than the naive scheme.

Cost of Query Processing We study the implication of data-item distributions on the cost of query processing. For the same size of queries, the cost of query processing in R-DHT is determined by the number of resource types (K). In

contrast, in conventional DHT, the cost is determined by the number of nodes (N). This indicates that (i) R-DHT is more suitable in applications where query selectivity is much larger than the number of query answers, and (ii) relaxing node autonomy through selective data-item distributions can improve the performance of multi-attribute range queries in R-DHT.

Resiliency to Node Failures We show that using R-DHT as the underlying overlay increases the resiliency of Midas without a need to replicate data items. Our simulation result shows that nearly all available keys are located even when 50% of nodes fail simultaneously.

Query Performance under Churn In R-DHT, effective backup fingers are crucial in increasing performance of query processing under churn. When there are less finger flexibility to exploit, we highlight hierarchical R-DHT as a possible solution where only stable nodes can be promoted into supernodes.

Chapter 5

Conclusion

We conclude this thesis by summarizing our main contributions and highlight several directions for future research to address the limitations of our proposed scheme.

5.1 Summary

We have proposed a DHT-based system that does not distribute data items across an overlay network. For a large distributed system consisting of many administrative domains, our proposed system addresses the issues of data-item ownership and conflicting self-interest among different administrative domains. Figure 5.1 summarizes our proposed scheme which consists of two main parts: R-DHT and Midas. R-DHT (***R***ead-only ***DHT***) is a new DHT abstraction that does not distribute data items across an overlay network. Two variants of R-DHT have been proposed, namely flat R-DHT and hierarchical R-DHT. In addition, we highlighted a hybrid scheme which allows *selective data-item distributions* in an R-DHT overlay network. Midas (***M***ulti***-d***imensional r***a***nge querie***s***) supports multi-attribute

range queries on R-DHT by exploiting a *d*-to-*one* mapping scheme. We have presented Midas's indexing multi-attribute resources and Midas's query engine. We have also demonstrated that in addition to R-DHT, Midas is applicable to support multi-attribute range queries on conventional DHT as well.

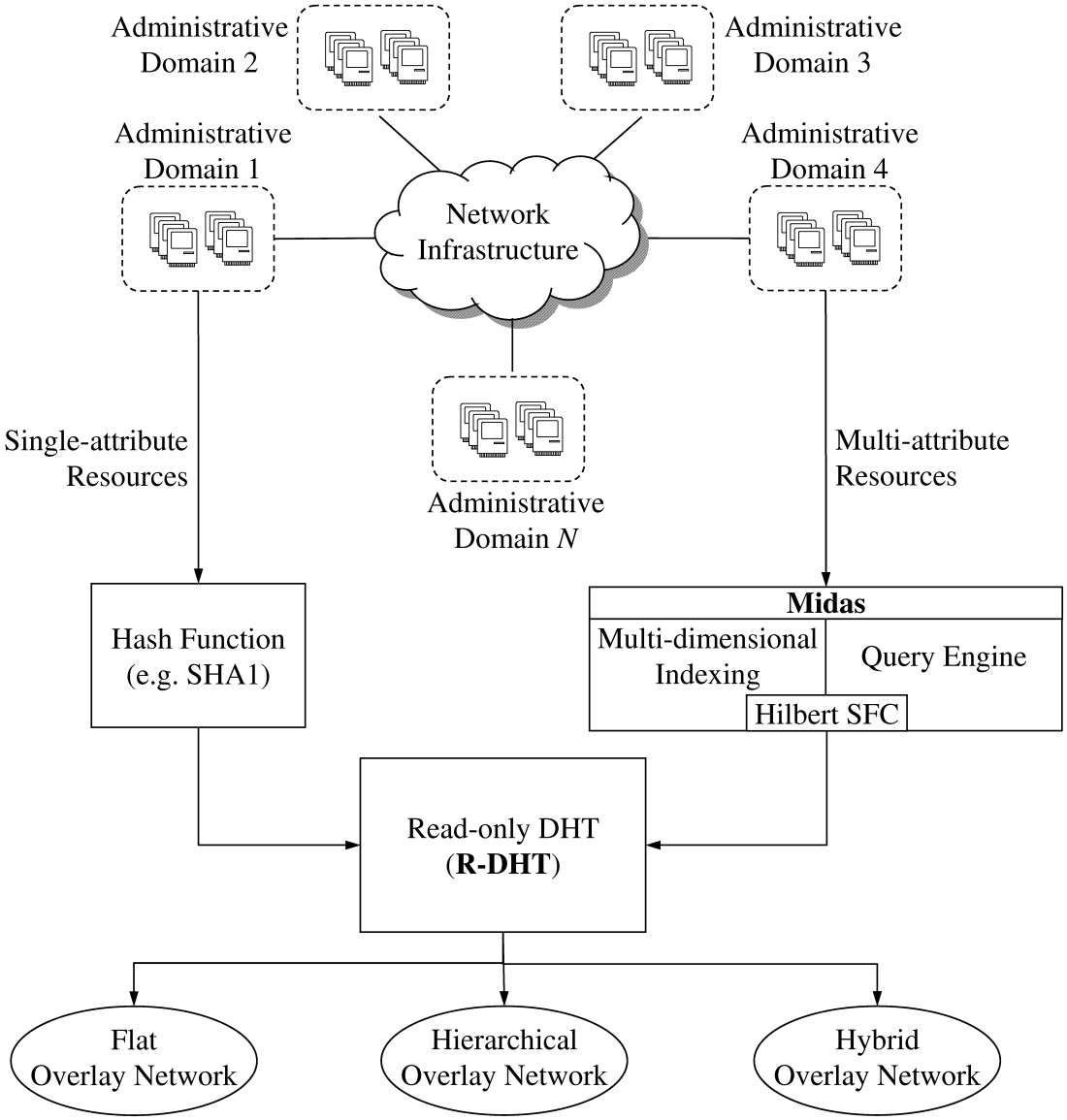


Figure 5.1: Multi-attribute Queries on R-DHT

In the following, we summarize our main contributions.

Effective and Efficient R-DHT Lookup

A significance result in this thesis is to demonstrate that even without distributing data items, the performance of R-DHT lookups is better than conventional DHT in two areas:

1. *Lookup Path Length*

   Although the size of R-DHT overlay is larger than conventional DHT, an R-DHT lookup is as efficient as a DHT lookup due to the three proposed optimizations, namely *routing by segments* and *shared finger tables*. Using Chord as the underlying overlay graph, we have shown that the lookup path length in Chord-based R-DHT is $O(\min(\log K, \log N))$ hops (i.e. better than Chord which is $O(\log N)$ hops), where N denotes the number of hosts and K denotes the number of unique keys (i.e. unique resource types). Our simulation results further confirm our theoretical analysis on R-DHT lookup path length.

2. *Lookup Resiliency to Node Failures*

   We have shown that R-DHT does not need to rely on active replication to achieve high resiliency. Firstly, failure of a node does not affect data items belonging to other nodes. Secondly, result of a lookup operation can be found in a segment which consists of multiple nodes. To achieve high lookup resiliency, we propose to exploit segment-based overlays through the *backup fingers* scheme. Through simulation experiments, we have demonstrated that both lookup operations and Midas query engine on R-DHT achieve a higher result guarantee: nearly all available keys are found even when half of nodes in an overlay network simultaneously fail.

Collision Detection and Resolution in Hierarchical R-DHT

To address the higher maintenance overhead in a flat overlay network, we proposed a hierarchical R-DHT, which collapses nodes within a segment into a second-level overlay network. We also addressed the problem of group collisions in hierarchical R-DHT. Collisions increase the size of the top-level overlay and in turn increase the maintenance cost of the top-level overlay. To detect collisions, we proposed a scheme whereby collision detections are performed together with stabilization to avoid introducing additional messages. To resolve collisions, we proposed a supernode-initiated and a node-initiated merging scheme.

Simulation analysis shows that our collision detection and resolution scheme is more effective when stabilization is performed more frequently. With our scheme, the number of collisions is reduced by 80% at least. In addition, the size of the top-level overlay remains close to the ideal size; otherwise it can be two to five times larger.

Support for Multi-Attribute Range Queries on R-DHT

We have proposed Midas, an approach to support multi-attribute range queries on R-DHT based on d-to-*one* mapping scheme. The selection of d-to-*one* as the basis of our solution, rather than distributed inverted index or d-to-d, is due to two main reasons: R-DHT does not distribute data items and there is more research interest, within the P2P community, on one-dimensional DHT compared to d-dimensional DHT.

We described the multi-dimensional indexing scheme in Midas which assigns to each multi-attribute resource a key derived using Hilbert SFC. We also proposed two optimizations for the query engine in Midas, namely *incremental search* and *search-key elimination*. Through simulation evaluations we have shown that Midas

significantly reduces the number of nodes visited in processing a query, compared to the naive R-DHT query processing.

Impact of Data-Item Distribution on Multi-Attribute Range Queries

We have studied the implication of data-item distribution to the performance of query processing. We compared the cost of query processing in two Midas implementations: one on R-DHT and another on conventional DHT. Simulation evaluations reveal two main observations:

1. *Data-item distribution reduces the number of nodes visited in processing a query.*

   The number of lookups per query is determined by the number of nodes that are responsible for query results. In conventional DHT, each node is a bucket with a number of unique keys, as opposed to R-DHT where each node is a bucket of one unique key. Thus, the number of responsible nodes in conventional DHT is lower than R-DHT. This reduces the query cost in conventional DHT.

2. *The higher cost of query processing in R-DHT is due to a higher number of R-DHT lookups required, not the cost of each individual R-DHT lookup.*

   Based on this observation, we have highlighted possible optimizations to reduce the cost of query processing in R-DHT, such as combining query processing with resource accesses and selective data-item distribution.

Performance of R-DHT under Churn

We have studied the performance of query processing in R-DHT under churn. Churn introduces inconsistencies to routing states maintained by nodes. From

our simulation evaluations, we have observed that R-DHT achieves a higher result guarantee when there are a higher number of nodes sharing resources of the same type. With a higher number of nodes within a segment, the exploitation of finger flexibility mitigates the impact of inconsistent node's routing states on the number of keys successfully retrieved. As such, the difference between R-DHT and conventional DHT is negligible. However, when each resource type is provisioned only by a small number of nodes, result guarantee in R-DHT is at most 20% worse than conventional DHT. Thus, R-DHT is suitable for a system whereby many administrative domains share resources of the same type, but administrative domains are willing to store only their own data items.

5.2 Future Works

We highlight several research directions for future work.

Selective Data-Item Distribution

An administrative domain who is willing to store data items belonging to other administrative domains joins R-DHT as one node only and occupies a reserved segment (see Appendix B for details). The benefits of selective data-item distribution include:

1. Selective data-item distribution improves the performance of multi-attribute range queries because data-items are aggregated on a trusted node. This reduces the number of lookups needed to retrieve all query result.

2. Selective data-item distribution facilitates replication of data items to a set of trusted nodes. This improves availability of resources as every resource is duplicated in different administrative domains. When the master copy a resource is unavailable, R-DHT can locate the backup copies of the resource.

3. Selective data-item distribution can address the load imbalance problem where all lookups for a frequently-requested data item are routed to its originating domain.

Further studies are needed to effectively exploit selective data-item distribution in R-DHT and to evaluate its impact on R-DHT performance.

Another interesting area to explore is how to selectively distribute data-item without reserving a reserved segment. The benefits of not requiring a reserved segment include:

1. In our current scheme, an R-DHT implementation has to provide two lookup interfaces: one to locate data items among read-only nodes, and another to locate data items among read-write nodes (in the reserved segments). Without a reserved segment, only one common lookup interface is required.

2. In our current scheme, when query results must be retrieved from both read-write and read-only nodes, each query is processed twice: once to retrieve keys from the reserved segment and once to retrieve keys from the remaining segments. By removing the reserved segment, we can retrieve keys from both set of nodes by processing each query only once.

A simple approach to remove the reserved segment is by exploiting host virtualizations. As illustrated in Figure 5.2, when host 5 stores a replicated version of key 5, a new node 5|3 is created. However, this simple approach increases the size of overlay network which in turns increases the maintenance overhead. Thus, a better approach is needed.

Dynamic Routing-Table Size

Virtualization in R-DHT increases the size of its overlay network in terms of number of nodes. We have proposed a hierarchical R-DHT scheme to partition

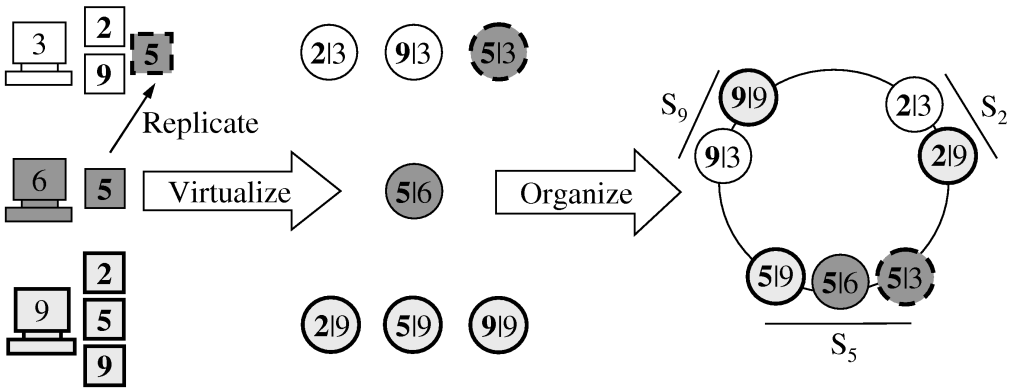


Figure 5.2: Exploiting Host Virtualization to Selectively Distribute Data Items

the maintenance overhead into multiple overlay networks. Assume that N denotes the number of hosts and K denotes the number of unique keys, the size of the top-level overlay in a hierarchical R-DHT is K groups and its maintenance cost is a function of K. However, when $K > N$, the maintenance cost can be further reduced into a function of N, i.e. the maintenance cost in conventional DHT. A possible solution to address this issue, which complements hierarchical R-DHT, is to investigate a new scheme where each administrative domain adaptively adjusts the number of fingers maintained. In this scheme, each administrative domain approximate the size of the overlay network to determine the minimum number finger required in order to support robust lookup with short lookup path length.

Caching of Data Items

Caching is a common technique to improve the lookup performance in DHT. From the perspective of nodes, caching data items belonging to other nodes is voluntarily, as opposed to data-item distribution which is mandatory. Thus, caching can be exploited to improve the lookup performance in R-DHT without violating the storage-usage policy of a node. However, we need to address the problem of data-item ownership due to malicious nodes. For example, a node can cache a data item indefinitely by ignoring invalidation requests from the owner of data items.

Semi-Structured Overlay Networks

R-DHT allows node autonomy in placing their key-value pairs. However, a higher degree of node autonomy is possible, in particular the autonomy in selecting neighboring nodes in an overlay network [44]. In DHT, the neighbors of a node is determined by a structure overlay network, i.e. the overlay network resembles a graph with a certain topology, and and the position of a node in an overlay network is determined by the node identifier (Section 1.2).

Recently, semi-structured overlay topologies have been proposed [37, 126]. In these schemes, nodes are free to choose its position in the overlay network and its neighbors, as long as the overlay network exhibits a global property such as a power-law network [126] or a square-root network [37]. Though both proposed schemes claim a provable lookup path length, we believe that more works are needed to improve these schemes. Firstly, percolation search [126] is based on earlier observations that file sharing P2P systems are power-law networks [94, 121]. However, a recent study indicates that such systems are not power-law [134]. Secondly, square-root network [37] assumes that the popularity of data items is known. However, the author does not describe how to measure the popularity of every data item in a large scale P2P system.

Appendix A

Read-Only CAN

In R-DHT, the identifier of a node is prefixed by the key shared by the node. However, this property is not guaranteed in CAN [116] because CAN dynamically changes the identifier of existing nodes when splitting a zone (Section 1.2.2). In this chapter, we describe two R-CAN schemes, flat R-CAN (see also Section 2.3.1) and hierarchical R-CAN (see also Section 3.2), whereby the dynamic zone splitting guarantees that a node identifier is prefixed by a key shared by a node. Flat R-CAN allows a zone to be split into two unequal-size subzones, whereas hierarchical R-CAN allows several nodes to occupy the same zone. Subsequently, we make a distinction between VIDs and node identifiers as stated in Definition A.1.

Definition A.1. *Let n denote a node identified by an m-bit node identifier. The i-bit VID of node n is defined as the i-bit prefix of n, where $i \leq m$.*

Based on Definition A.1, an m-bit node identifier is associated with m VIDs. Figure A.1 illustrates four possible VIDs of a 4-bit node identifier.

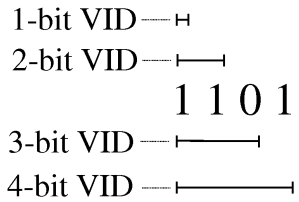
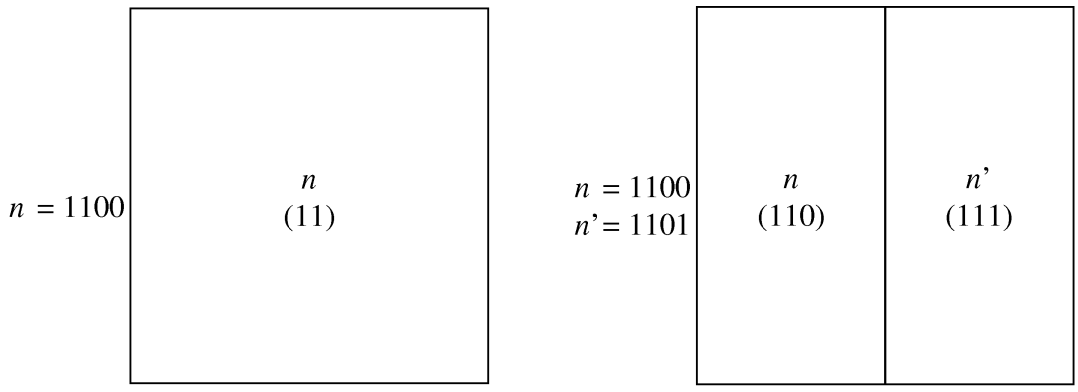


Figure A.1: VIDs of Node Identifier 1101_2

A.1 Flat R-CAN

The original CAN dynamically changes the location of nodes and as such, the i-bit VID of a node may not be the same as the i-bit prefix of the node identifier. As illustrated in Figure A.2, when node n' arrives, a zone, which is occupied by node n only, is split along x-dimension into two *equal-size* subzones; each node is assigned one subzone. Nodes reflect their new subzone in their 3-bit VID, i.e. the 2-bit VID of n (before the splitting) concatenated by 0 or 1. However, this new VID for n' violates Definition A.1 because it does not match with the 3-bit prefix of n', which is 111.



(a) Zone Occupied by Node n whose 2-bit VID is 11

(b) 3-bit VID Assigned to n' is not Equal to 3-bit Prefix of n'

Figure A.2: Zone Splitting in CAN may Violate Definition A.1

Flat R-CAN solves the above problems as follows:

1. A node's location in the Cartesian space is determined only by its (fixed)

node identifier.

This is in contrast to the original CAN, where a new node chooses a random initial location.

2. A zone can be split into *unequal-size* subzones.

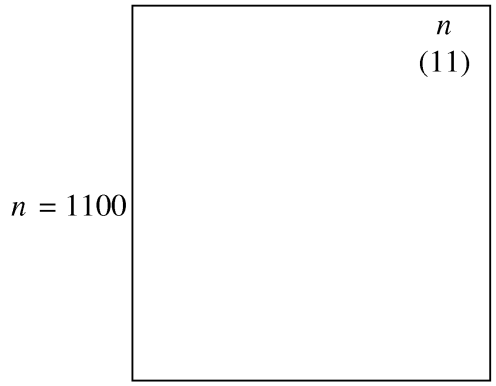
3. We ignore a zone splitting along a particular dimension two nodes, i.e. one existing node and one new node, occupy the same coordinate in the splitting dimension.

   When a zone splitting is ignored, we assigned to nodes their i-bit node-identifier as their i-bit VID. Then, we continue to split the zone along a different dimension until the i'-bit VID produced by the splitting, where $i' > i$, is equal to the i'-bit prefix of a node identifier.

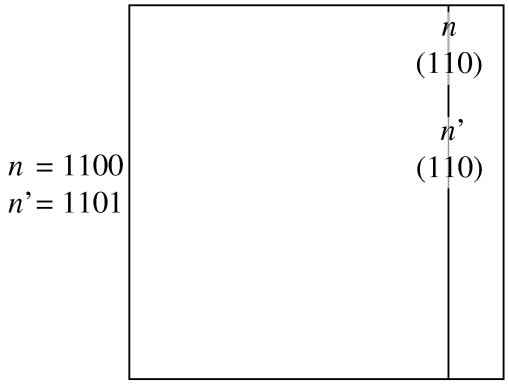
As illustrated in Figure A.3a, the location occupied by node n is determined by its node identifier. When node n' with the same x-coordinate as n enters the zone, splitting the zone along x-dimension is ignored (Figure A.3b). Thus, both nodes are a 3-bit VID derived from their 3-bit node-identifier prefix. Afterwards, we split the zone along y-dimension and assign a 4-bit VID to each node (Figure A.3c). Because n and n' have different a y-coordinate, two 4-bit VIDs are produced; these VIDs are guaranteed to equal to the 4-bit prefix of n and n'.

Theorem A.1. *Let node identifiers are m-bit long. The number of ignored zone splittings is at most $m - 1$.*

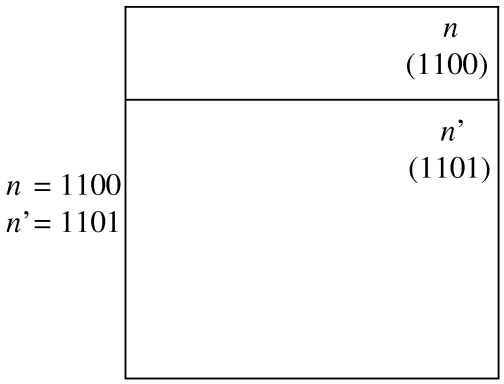
Proof. Assume that two node identifiers share the same $(m - 1)$-bit long prefix, i.e. they differ in the least-significant bit only. Then, out of m VID owned by each node, $(m - 1)$ VID are shared between nodes. In R-CAN, the zone-splitting process ignores the splittings that result in the same VID for both nodes. Thus, $m - 1$ splitting are ignored.



(a) Node Identifier Determines Coordinates of a Node



(b) Zone Splitting along x-Dimension is Ignored



(c) Continue Zone Splitting along y-Dimension

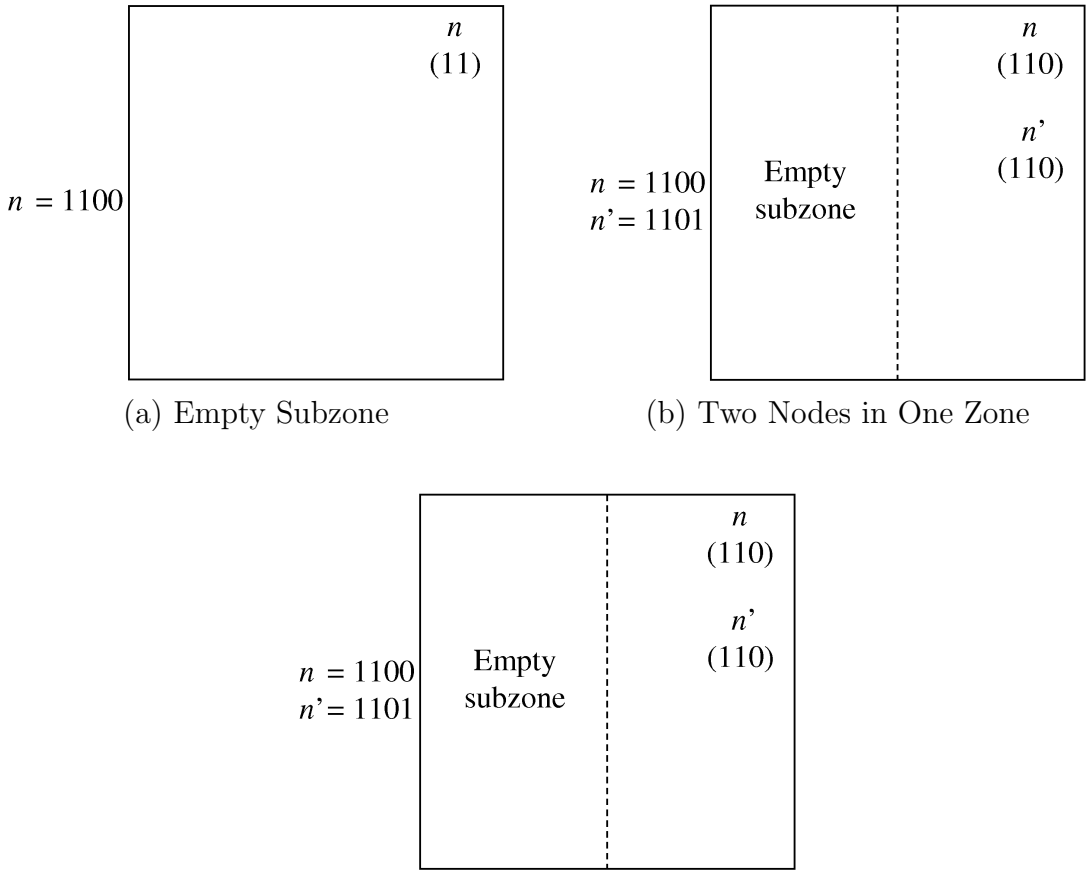
Figure A.3: Zone Splitting in Flat R-CAN

In other words, given any two nodes, their coordinates must differ in at least one dimension. When we split a zone, we ignore the splittings along dimensions in which both nodes have the same coordinate. □

A.2 Hierarchical R-CAN

The primary difference between hierarchical R-CAN and flat R-CAN is that in hierarchical R-CAN, a zone is either split to two *equal-size* subzones or none at all. As illustrated in Figure A.4, splitting the zone along x-dimension into two equal-size subzones results in an empty subzone. However, unlike flat R-CAN which selects an alternative splitting dimension, hierarchical R-CAN simply does

not split the zone. Instead, the zone is shared by both the original node n and the new node n'. Thus, this zone can be considered as a group as in hierarchical DHT. Nodes within this zone may be further organized as a second-level overlay network.



(a) Empty Subzone

(b) Two Nodes in One Zone

(c) Two Nodes in One Zone

Figure A.4: Zone Splitting in Hierarchical R-CAN

Appendix B

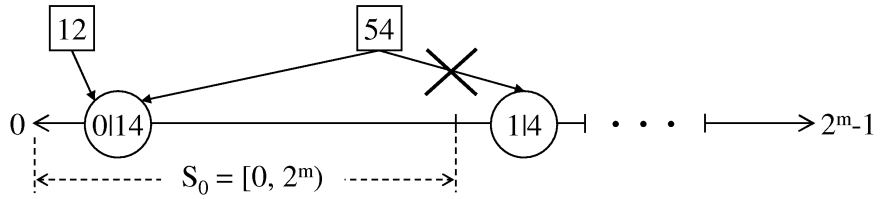
Selective Data-Item Distribution

In Chapter 2, we have demonstrated how R-DHT supports node autonomy where each node stores only its own data items. In this chapter, we extend R-DHT to accommodate applications where some hosts may store data items belonging to other hosts. Example of such hosts are DHT service providers [23] or MDS servers [4] serving as yellow pages in a computational grid. Selective data-item distribution also facilitates data-item replication in R-DHT.

To accommodate publicly-writable hosts, R-DHT restricts data-item distribution within a reserved segment S_r (e.g. r could be 0 or $2^m - 1$). A publicly-writable host (h) is virtualized into only one node (n) identified with $r|h$. A key is then mapped and stored onto a node within S_r even if another node outside S_r is the closest to the key. For example, R-Chord maps and stores key k onto node $n = r|h$ where $r|h = successor(r|k)$; this can be further simplified as mapping key k to publicly-writable host h where $h = successor(k)$. Essentially, the selective data-item distribution scheme emulates an m-bit node-identifier space within the $2m$-bit identifier space. Our selective data-item distribution reduces the maintenance

overhead of R-DHT because each publicly-writable host increases the size of the overlay network only by one node.

Figure B.1b shows the algorithm for a publicly-writable host joining the reserved segment.



(a) Map Keys to Nodes only in Segment S_0

```
1.  // Host h joins segment S_r
2.  // through an existing host e.
3.  h.virtualize_to_reserve_segment(e)
4.      n = r|h;
5.      n.join(e)   // Chord's protocol [133]
```

(b) Virtualize Host to Reserved Segment

Figure B.1: Relaxing Node Autonomy

Figure B.2 shows the algorithm for finding $successor(k)$ in segment S_r. This operation allows the mapping of a key onto a node in S_r (i.e. store operation) and the retrieval of a key from segment S_r (i.e. lookup operation). The algorithm first finds the reserved segment S_r if necessary (line 5), followed by finding $successor(k)$ in S_r (line 16 and 22). If no such node is found, i.e. k is beyond the last node in S_r, R-Chord maps k onto $successor(r|0)$, i.e. the first node in S_r (line 14 and 20).

```
1.   // Find successor(k) in segment S_r
2.   h.find_successor_in_rsegment(k)
3.       n = r|h;
4.
5.       if ∄n then
6.           // Find S_r, as h is not publicly writable
7.           h' = lookup(r);
8.           if h' == NOT_FOUND then
9.               return NOT_FOUND;
10.          return h'.find_successor_in_rsegment(k);
11.
12.      if n < r|k ≤ n.successor then
13.          // n is the predecessor of successor(k)
14.          if prefix(n.successor) == r then
15.              return n.successor;
16.          return find_successor(r|0);    // See Figure 2.18b
17.
18.      // Go to the nearest known predecessor of k
19.      n' = closest_preceding_node(r|k)
20.      if prefix(n') == r then
21.          return n'.find_successor_in_segment(k);
22.      return n'.find_successor(r|0);    // See Figure 2.18b
```

Figure B.2: Lookup within Reserved Segment

References

[1] Apache HTTP server. http://httpd.apache.org.

[2] The Chord Project. http://www.pdos.lcs.mit.edu/chord.

[3] EarthLink SIPShare. http://www.research.earthlink.net/p2p/.

[4] Globus toolkit – Information Service. http://www.globus.org/toolkit/mds/.

[5] GLUE information model. http://glueschema.forge.cnaf.infn.it.

[6] Gnutella. http://www.gnutella.com.

[7] IEEE standard 1420.1-1995 (R2002), IEEE standard for information technology–software reuse–data model for reuse library interoperability: Basic interoperability data model (BIDM). http://standards.ieee.org/reading/ieee/std/se/1420.1-1995.pdf.

[8] Napster. http://www.napster.com.

[9] Oracle Spatial. http://www.oracle.com/technology/products/spatial/index.html.

[10] P2Pwg: Peer-to-peer working group. http://p2p.internet2.edu.

[11] Qnext. http://www.qnext.com.

[12] Skype. http://www.skype.com.

[13] The voP2P project. http://vop2p.jxta.org.

[14] K. Aberer, P. Cudr-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt. P-Grid: A self-organizing structured p2p system. *SIGMOD Record*, 32(2):29–33, September 2003.

[15] L. A. Adamic, R. M. Lukose, A. R. Puniyani, and B. A. Huberman. Local search in unstructured networks. *Handbook of Graphs and Networks: From the Genome to the Internet*, pp. 295–316, January 2003.

[16] D. Agrawal, A. E. Abbadi, and S. Suri. Attribute-based access to distributed data over P2P networks. *Proc. of the 4th Intl. Workshop on Databases in Networked Information Systems*, pp. 244–263, Springer-Verlag, Japan, March 2005.

[17] L. O. Alima, S. El-Ansary, P. Brand, and S. Haridi. DKS (N, k, f): A family of low communication, scalable and fault-tolerant infrastructures for P2P applications. *Proc. of the 3rd IEEE Intl. Symp. on Cluster Computing and the Grid*, pp. 344–350, IEEE Computer Society Press, Japan, May 2003.

[18] S. Androutsellis-Theotokis and D. Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys*, 36(4):335–371, December 2004.

[19] A. Andrzejak and Z. Xu. Scalable, efficient range queries for grid information services. *Proc. of the 2nd Intl. Conf. on Peer-to-Peer Computing*, pp. 33–40, IEEE Computer Society Press, Sweden, September 2002.

[20] J. Aspnes and G. Shah. Skip Graphs. *Proc. of the 14th Annual ACM-SIAM Symp. on Discrete Algorithms*, pp. 384–393, ACM/SIAM Press, USA, January 2003.

[21] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, 1999.

[22] H. Balakrishnan, K. Lakshminarayanan, S. Ratnasamy, S. Shenker, I. Stoica, and M. Walfish. A layered naming architecture for the internet. *Proc. of ACM SIGCOMM*, pp. 343–352, ACM Press, Germany, September 2004.

[23] H. Balakrishnan, S. Shenker, and M. Walfish. Peering peer-to-peer providers. *Proc. of the 4th Intl. Workshop on Peer-to-Peer Systems*, pp. 104–114, Springer-Verlag, USA, February 2005.

[24] D. Bauer, P. Hurley, R. Pletka, and M. Waldvogel. Bringing efficient advanced queries to distributed hash tables. *Proc. of the 29th IEEE Intl. Conf. on Local Computer Networks*, pp. 6–14, IEEE Computer Society Press, USA, November 2004.

[25] R. Bhagwan, S. Savage, and G. M. Voelker. Understanding availability. *Proc. of the 2nd Intl. Workshop on Peer-to-Peer Systems*, pp. 256–267, Springer-Verlag, USA, February 2003.

[26] G. Breinholt and C. Schierz. Algorithm 781: Generating Hilbert's space-filling curve by recursion. *ACM Transactions on Mathematical Software*, 24(2):184–189, June 1998.

[27] A. R. Butt, R. Zhang, and Y. C. Hu. A self-organizing flock of Condors. *Proc. of the ACM/IEEE SC2003 Conf. on High Performance Networking and Computing*, pp. 42, ACM Press, USA, November 2003.

[28] M. Cai, M. Frank, J. Chen, and P. Szekely. MAAN: A Multi-Attribute Addressable Network for grid information services. *Journal of Grid Computing*, 2(1):3–14, 2004.

[29] M. Castro, M. Costa, and A. Rowstron. Performance and dependability of structured peer-to-peer overlays. *Proc. of the 2004 Intl. Conf. on Dependable Systems and Networks*, pp. 9–18, June 2004.

[30] M. Castro, M. Costa, and A. Rowstron. Should we build Gnutella on a structured overlay? *ACM SIGCOMM Computer Communication Review*, 34(2):131–136, April 2004.

[31] M. Castro, M. Costa, and A. Rowstron. Debunking some myths about structured and unstructured overlays. *Proc. of 2nd Symp. on Networked Systems Design and Implementation*, pp. 85–98, USENIX Association, USA, May 2005.

[32] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Secure routing for structured peer-to-peer overlay networks. *Proc. of the 5th USENIX Symp. on Operating Systems Design and Implementation*, pp. 299–314, USENIX Association, USA, December 2002.

[33] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making Gnutella-like P2P systems scalable. *Proc. of ACM SIGCOMM*, pp. 407–418, ACM Press, Germany, August 2003.

[34] F. Chen, T. Repantis, and V. Kalogeraki. Coordinated media streaming and transcoding in peer-to-peer system. *Proc. of the 19th IEEE Intl. Parallel and Distributed Processing Symp.*, pp. 56b, IEEE Computer Society Press, USA, April 2005.

[35] A-H. Cheng and Y-J. Joung. Probabilistic file indexing and searching in unstructured peer-to-peer networks. *Proc. of the 4th IEEE Intl. Symp. on Cluster Computing and the Grid*, pp. 9–18, IEEE Computer Society Press, USA, April 2004.

[36] A. J. Cole. Compaction techniques for raster scan graphics using space-filling curves. *The Computer Journal*, 31(1):87–92, 1987.

[37] B. F. Cooper. Quickly routing searches without having to move content. *Proc. of the 4th Intl. Workshop on Peer-to-Peer Systems*, pp. 163–172, Springer-Verlag, USA, February 2005.

[38] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making backup cheap and easy. *Proc. of the 5th USENIX Symp. on Operating Systems Design and Implementation*, pp. 285–298, USENIX Association, USA, December 2002.

[39] L. P. Cox and B. D. Noble. Samsara: Honor among thieves in peer-to-peer storage. *Proc. of the 19th ACM Symp. on Operating Systems Principles*, pp. 120–132, ACM Press, USA, October 2003.

[40] A. Crespo and H. Garcia-Molina. Routing indices for peer-to-peer systems. *Proc. of the 22nd IEEE Intl. Conf. On Distributed Computing Systems*, pp. 23–33, IEEE Computer Society Press, Austria, July 2002.

[41] Y. Cui and K. Nahrstedt. Layered peer-to-peer streaming. *Proc. of 13th Intl. Workshop on Network and Operating Systems Support for Digital Audio and Video*, pp. 162–171, ACM Press, USA, June 2003.

[42] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. *Proc. of the 11th ACM Symp. on Operating Systems Principles*, pp. 202–215, ACM Press, Canada, October 2001.

[43] F. Dabek, B. Y. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a common API for structured peer-to-peer overlays. *Proc. of the 2nd Intl. Workshop on Peer-to-Peer Systems*, pp. 33–44, Springer-Verlag, USA, February 2003.

[44] N. Daswani, H. Garcia-Molina, and B. Yang. Open problems in data-sharing peer-to-peer systems. *Proc. of the 9th Intl. Conf. on Database Theory*, pp. 1–15, Springer-Verlang, Italy, January 2003.

[45] F. K. H. A. Dehne, T. Eavis, and A. Rau-Chaplin. Parallel multi-dimensional ROLAP indexing. *Proc. of the 3rd Intl. Symp. on Cluster Computing and the Grid*, pp. 86–95, IEEE Computer Society Press, Japan, May 2003.

[46] P. Druschel and A. I. T. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. *Proc. of the 8th Workshop on Hot Topics in Operating Systems*, pp. 75–80, IEEE Computer Society Press, Germany, May 2001.

[47] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 1999.

[48] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with Coral. *Proc. of the 1st Symp. on Networked Systems Design and Implementation*, pp. 239–252, USENIX Association, USA, March 2004.

[49] V. Gaede and O. Günther. Multidimensional access methods. volume 30, pp. 170–231, June 1998.

[50] P. Ganesan, B. Yang, and H. Garcia-Molina. One torus to rule them all: Multi-dimensional queries in P2P systems. *Proc. of the 7th Intl. Workshop on the Web and Databases*, pp. 19–24, France, June 2004.

[51] L. Garcés-Erice, E. W. Biersack, P. A. Felber, K. W. Ross, and G. Urvoy-Keller. Hierarchical peer-to-peer systems. *Proc. of the 9th Intl. Euro-Par Conf.*, pp. 1230–1239, Springer-Verlag, Austria, August 2003.

[52] G. Ghinita and Y. M. Teo. An adaptive stabilization framework for distributed hash tables. *Proc. of the 20th IEEE Intl. Parallel and Distributed Processing Symp.*, IEEE Computer Society Press, Greece, April 2006.

[53] A. Ghodsi, L. O. Alima, and S. Haridi. Low-bandwidth topology maintenance for robustness in structured overlay networks. *Proc. of 38th Hawaii Intl. Conf. on System Sciences*, pp. 302a, IEEE Computer Society Press, USA, January 2005.

[54] A. Ghodsi, L. O. Alima, and S. Haridi. Symmetric replication for structured peer-to-peer systems. *Proc. of the 3rd Intl. Workshop on Databases, Information Systems and Peer-to-Peer Computing*, pp. 12, Spinger-Verlag, Norway, April 2005.

[55] The Boston Globe. Google subpoena roils the web: US effort raises privacy issues. `http://www.boston.com/news/nation/articles/2006/01/21/google_subpoena_roils_the_web?mode=PF`, January 2006.

[56] O. D. Gnawali. A keyword-set search system for peer-to-peer networks. Master's thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 2002.

[57] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in dynamic structured P2P systems. *Proc. of INFOCOM*, pp. 2253– 2262, IEEE Press, China, March 2004.

[58] P. B. Godfrey and I. Stoica. Heterogeneity and load balance in distributed hash tables. *Proc. of INFOCOM*, pp. 596–606, IEEE Press, USA, March 2005.

[59] Google. Google's opposition to the government's motion to compel. `http://googleblog.blogspot.com/pdf/Google_Oppo_to_Motion.pdf`, February 2006.

[60] Google. Response to the DoJ motion. `http://googleblog.blogspot.com/2006/02/response-to-doj-motion.html`, February 2006.

[61] C. Gotsman and M. Lindenbaum. On the metric properties of discrete space-filling curves. *IEEE Transactions on Image Processing*, 5(5):794–797, May 1996.

[62] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of DHT routing geometry on resilience and proximity. *Proc. of ACM SIGCOMM*, pp. 381–394, ACM Press, Germany, August 2003.

[63] P. K. Gummadi, S. Saroiu, and S. Gribble. Measurement study of Napster and Gnutella as examples of peer-to-peer file sharing systems. *Multimedia Systems Journal*, 9(2):170–184, August 2003.

[64] A. Gupta, B. Liskov, and R. Rodrigues. Efficient routing for peer-to-peer overlays. *Proc. of 1st Symp. on Networked Systems Design and Implementation*, pp. 113–126, USENIX Association, USA, March 2004.

[65] I. Gupta, K. Birman, P. Linga, A. Demers, and R. V. Renesse. Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. *Proc. of the 2nd Intl. Workshop on Peer-to-Peer Systems*, pp. 160–169, Springer-Verlag, USA, February 2003.

[66] Andreas Haeberlen, Alan Mislove, and Peter Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. *Proc. of 2nd Symp. on Networked Systems Design and Implementation*, pp. 143–158, USENIX Association, USA, May 2005.

[67] M. Harren, J. M. Hellerstein, R. Huebsch, B.T.Loo, S. Shenker, and I. Stoica. Complex queries in DHT-based peer-to-peer networks. *Proc. of the 1st Intl. Workshop on Peer-to-Peer Systems*, pp. 242–249, Springer-Verlag, USA, March 2002.

[68] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A scalable overlay network with practical locality properties. *Proc. of the 4th USENIX Symp. on Internet Technologies and Systems*, pp. 113–126, USENIX Association, USA, March 2003.

[69] H-C. Hsiao and C-T. King. A tree model for structured peer-to-peer protocols. *Proc. of the 3rd IEEE Intl. Symp. on Cluster Computing and the Grid*, pp. 336–343, IEEE Computer Society Press, Japan, May 2003.

[70] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the internet with PIER. *Proc. of 29th Intl. Conf. on Very Large Data Bases*, pp. 321–332, Morgan Kaufmann Publishers, Germany, September 2003.

[71] A. Iamnitchi. *Resource Discovery in Large Resource-Sharing Environments*. PhD thesis, Dept. of Computer Science, The University of Chicago, December 2003.

[72] A. Iamnitchi, M. Ripeanu, and I. Foster. Locating data in (small-world?) peer-to-peer scientific collaborations. *Proc. of the 1st International Workshop on Peer-to-Peer Systems*, pp. 232–241, Springer-Verlag, USA, March 2002.

[73] S. Iyer, A. I. T. Rowstron, and P. Druschel. Squirrel: a decentralized peer-to-peer web cache. *Proc. of the 21th ACM Symp. on Principles of Distributed Computing*, pp. 213–222, ACM Press, USA, July 2002.

[74] H. V. Jagadish. Linear clustering of objects with multiple attributes. *Proc. of the 1990 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 332–342, ACM Press, USA, June 1990.

[75] G. Jin and J. Mellor-Crummey. SFCGen: A framework for efficient generation of multi-dimensional space-filling curve by recursion. *ACM Transactions on Mathematical Software*, 31(1):120–148, March 2005.

[76] M. F. Kaashoek and D. R. Karger. Koorde: A simple degree-optimal distributed hash table. *Proc. of the 2nd Intl. Workshop on Peer-to-Peer Systems*, pp. 98–107, Springer-Verlag, USA, February 2003.

[77] D. R. Karger and M. Ruhl. Diminished Chord: A protocol for heterogeneous subgroup. *Proc. of the 3rd Intl. Workshop on Peer-to-Peer Systems*, pp. 288–297, Springer-Verlag, USA, February 2004.

[78] D. R. Karger and M. Ruhl. Simple, efficient load balancing algorithms for peer-to-peer systems. *Proc. of the 3rd Intl. Workshop on Peer-to-Peer Systems*, pp. 131–140, Springer-Verlag, USA, February 2004.

[79] F. B. Kashani and C. Shahabi. Criticality-based analysis and design of unstructured peer-to-peer networks as "complex systems". *Proc. of the 3rd IEEE Intl. Symp. on Cluster Computing and the Grid*, pp. 351–358, IEEE Computer Society Press, Japan, May 2003.

[80] K. Krauter, R. Buyya, and M. Maheswaran. A taxonomy and survey of grid resource management systems for distributed computing. *Intl. Journal of Software, Practice and Experience*, 32(2):135–164, February 2002.

[81] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. *Proc. of the 9th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 190–201, ACM Press, USA, November 2000.

[82] J. B. Kwon and H. Y. Yeom. Distributed multimedia streaming over peer-to-peer networks. *Proc. of the 9th Intl. Euro-Par Conf.*, pp. 851–858, Springer-Verlag, Austria, August 2003.

[83] M. Landers, H. Zhang, and K-L. Tan. Peerstore: Better performance by relaxing in peer-to-peer backup. *Proc. of the 4th Intl. Conf. on Peer-to-Peer Computing*, pp. 72–79, IEEE Computer Society Press, Switzerland, August 2004.

[84] J. K. Lawder. Using state diagrams for Hilbert curve mappings. Technical Report JL2/00, School of Computer Science and Information Systems, Birkbeck College, University of London, August 2000.

[85] J. K. Lawder and P. J. H. King. Using space-filling curves for multi-dimensional indexing. *Proc. of the 17th British National Conf. on Databases: Advances in Databases*, pp. 20–35, Springer-Verlag, UK, July 2000.

[86] J. Lee, H. Lee, S. Kang, S. Choe, and J. Song. CISS: An efficient object clustering framework for DHT-based peer-to-peer applications. *Proc. of VLDB Workshop On Databases, Information Systems and Peer-to-Peer Computing*, pp. 215–229, Spinger-Verlag, Canada, August 2004.

[87] M. Leslie, J. Davies, and T. Huffman. Replication strategies for reliable decentralised storage. *Proc. of the 1st Workshop on Dependable and Sustainable Peer-to-Peer Systems*, pp. 740–747, IEEE Computer Society Press, Japan, April 2006.

[88] J. Li, P. A. Chou, and C. Zhang. Mutualcast: An efficient mechanism for content distribution in a peer-to-peer (P2P) network. Technical Report MSR-TR-2004-100, Microsoft Research, Communication and Collaboration Systems, September 2004.

[89] J. Li, J. Stribling, T. M. Gil, R. Morris, and M. F. Kaashoek. Comparing the performance of distributed hash tables under churn. *Proc. of the 3rd Intl. Workshop on Peer-to-Peer Systems*, pp. 87–99, Springer-Verlag, USA, February 2004.

[90] J. Li, J. Stribling, R. Morris, and M. F. Kaashoek. Bandwidth-efficient management of DHT routing tables. *Proc. of 2nd Symp. on Networked Systems Design and Implementation*, pp. 99–114, USENIX Association, USA, May 2005.

[91] W. Li, Z. Xu, F. Dong, and J. Zhang. Grid resource discovery based on a routing-transferring model. *Proc. of the 3rd Intl. Workshop on Grid Computing*, pp. 145–156, Springer-Verlag, USA, November 2002.

[92] X. Liu and G. Schrack. Encoding and decoding the Hilbert order. *Software—Practice and Experience*, 26(12):1335–1346, December 1996.

[93] B. T. Loo, R. Huebsch, I. Stoica, and J. M. Hellerstein. The case for a hybrid P2P search infrastructure. *Proc. of the 3rd Intl. Workshop on Peer-to-Peer Systems*, pp. 141–150, Springer-Verlag, USA, February 2004.

[94] Q. Lv, P Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. *Proc. of the 2002 Intl. Conf. on Supercomputing*, pp. 84–95, ACM Press, USA, June 2002.

[95] V. March and Y. M. Teo. Multi-attribute range queries on read-only DHT. *Proc. of the 15th Intl. Conf. on Computer Communications and Networks*, pp. 419–424, IEEE Communications Society Press, USA, October 2006.

[96] V. March, Y. M. Teo, H. B. Lim, P. Eriksson, and R. Ayani. Collision detection and resolution in hierarchical peer-to-peer systems. *Proc. of the 30th IEEE Conf. on Local Computer Networks*, pp. 2–9, IEEE Computer Society Press, Australia, November 2005.

[97] V. March, Y. M. Teo, and X. Wang. DGRID: A DHT-based grid resource indexing and discovery scheme for computational grids. *Proc. of the 5th Australasian Symp. on Grid computing and e-Research*, pp. 41–48, Australian Computer Society Inc., Australia, January 2007.

[98] E. P. Markatos. Tracing a large-scale peer to peer system: An hour in the life of Gnutella. *Proc. of the 2nd IEEE Intl. Symp. on Cluster Computing and the Grid*, pp. 65–74, IEEE Computer Society Press, Germany, May 2002.

[99] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. *Proc. of the 1st Intl. Workshop on Peer-to-Peer Systems*, pp. 53–65, Springer-Verlag, USA, March 2002.

[100] D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pryune, B. Richard, S. Rollins, and Z. Xu. Peer-to-peer computing. Technical Report HPL-2002-57, HP Laboratories Palo Alto, March 2002.

[101] A. Mislove and P. Druschel. Providing administrative control and autonomy in structured peer-to-peer overlays. *Proc. of the 3rd Intl. Workshop on Peer-to-Peer Systems*, pp. 162–172, Springer-Verlag, USA, February 2004.

[102] A. Mislove, A. Post, C. Reis, P. Willmann, P. Druschel, D. S. Wallach, X. Bonnaire, P. Sens, J-M. Busca, and L. B. Arantes. POST: A secure, resilient, cooperative messaging system. *Proc. of the 9th Workshop on Hot Topics in Operating Systems*, pp. 61–66, IEEE Computer Society Press, USA, May 2003.

[103] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis on the clustering properties of the Hilbert space-filling curve. *IEEE Transactions on Knowledge and Data Engineering*, 13(1):124–141, January 2001.

[104] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. *Proc. of 5th USENIX Symp. on Operating System Design and Implementation*, USENIX Association, USA, December 2002.

[105] K. Nakauchi, Y. Ishikawa, H. Morikawa, and T. Aoyama. Peer-to-peer keyword search using keyword relationship. *Proc. of the 3rd IEEE Intl. Symp. on Cluster Computing and the Grid*, pp. 359–366, IEEE Computer Society Press, Japan, May 2003.

[106] Z. Németh and V. Sunderam. Characterizing grids: Attributes, definitions, and formalisms. *Journal of Grid Computing*, 1(1):9–23, 2003.

[107] World Association of Newspapers. Google must pay! `http://www.wan-press.org/article9384.html?var_recherche=google+news`.

[108] World Association of Newspapers. Newspaper, magazine and book publishers organizations to address search engine practices. `http://www.wan-press.org/article9055.html`, January 2006.

[109] F. D. Ngoc, J. Keller, and G. Simon. MAAY: a decentralized personalized search system. *Proc. of the IEEE/IPSJ Intl. Symp. on Applications and the Internet*, IEEE Computer Society Press, USA, January 2006.

[110] S. J. Nielson, S. A. Crosby, and D. S. Wallach. A taxonomy of rational attacks. *Proc. of the 4th Intl. Workshop on Peer-to-Peer Systems*, pp. 36–46, Springer-Verlag, USA, February 2005.

[111] B. C. Ooi, Y. Shu, and K.L. Tan. Relational data sharing in peer-based data management systems. *SIGMOD Record*, 32(1):59–64, March 2003.

[112] A. Oram. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. O'Reilly, 2001.

[113] V. Ramasubramanian and E. G. Sirer. Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays. *Proc. of 1st Symp. on Networked Systems Design and Implementation*, pp. 99–112, USENIX Association, USA, March 2004.

[114] L. Ramaswamy, B. Gedik, and L. Liu. A distributed approach to node clustering in decentralized peer-to-peer networks. *IEEE Transaction on Parallel and Distributed Systems*, 16(9):814–829, September 2005.

[115] F. Ramsak, V. Markl, R. Fenk, M. Zirkel, K. Elhardt, and R. Bayer. Integrating the UB-tree into a database system kernel. *Proc. of 26th Intl. Conf. on Very Large Data Bases*, pp. 263–272, Morgan Kaufmann Publishers, Egypt, September 2000.

[116] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable Content-Addressable Network. *Proc. of ACM SIGCOMM*, pp. 161–172, ACM Press, USA, August 2001.

[117] S. Ratnasamy, I. Stoica, and S. Shenker. Routing algorithms for DHTs: Some open questions. *Proc. the 1st Intl. Workshop on Peer-to-Peer Systems*, pp. 45–52, Springer-Verlag, USA, March 2002.

[118] Reuters. WPP's Sorrell sees Google as threat, opportunity. `http://today.reuters.com/news/articlebusiness.aspx?type=media&storyid=nN01402884&imageid=&cap=`, March 2006.

[119] S. Rhea, D. Geels, and T. Roscoe J. Kubiatowicz. Handling churn in a DHT. *Proc. of the USENIX*, pp. 127–140, USENIX Association, USA, June 2004.

[120] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A public DHT service and its uses. *Proc. of ACM SIGCOMM*, pp. 73–84, ACM Press, USA, August 2005.

[121] M. Ripeanu, I. Foster, and A. Iamnitchi. Mapping the Gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. *IEEE Internet Computing Journal*, 6(1):50–57, January 2002.

[122] R. Rodrigues and C. Blake. When multi-hop peer-to-peer lookup matters. *Proc. of the 3rd Intl. Workshop on Peer-to-Peer Systems*, pp. 112–122, Springer-Verlag, USA, February 2004.

[123] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. *Proc. of IFIP/ACM Intl. Conf. on Distributed Systems Platforms*, pp. 329–350, Springer-Verlag, Germany, November 2001.

[124] H. Sagan. *Space-Filling Curves*. Springer-Verlag, 1999.

[125] D. Sandler, A. Mislove, A. Post, and P. Druschel. FeedTree: Sharing micronews with peer-to-peer event notification. *Proc. of the 4th Intl. Workshop on Peer-to-Peer Systems*, pp. 141–151, Springer-Verlag, USA, February 2005.

[126] N. Sarshar, P. O. Boykin, and V. P. Roychowdhury. Percolation search in power law networks: Making unstructured peer-to-peer networks scalable. *Proc. of the 4th Intl. Conf. on Peer-to-Peer Computing*, pp. 2–9, IEEE Computer Society Press, Switzerland, August 2004.

[127] C. Schmidt and M. Parashar. Flexible information discovery in decentralized distributed systems. *Proc. of the 12th IEEE Intl. Symp. on High Performance Distributed Computing*, pp. 226–235, IEEE Computer Society Press, USA, June 2003.

[128] C. Schmidt and M. Parashar. Analyzing the search characteristics of space filling curve-based indexing within the Squid P2P data discovery system. Technical Report TR-276, Center for Advanced Information Processing (CAIP), Rutgers University, December 2004.

[129] S. Shi, G. Yang, D. Wang, J. Yu, S. Qu, and M. Chen. Making peer-to-peer keyword searching feasible using multi-level partitioning. *Proc. of the 3rd Intl. Workshop on Peer-to-Peer Systems*, pp. 151–161, Springer-Verlag, USA, February 2004.

[130] J. Shneidman and D. C. Parkes. Rationality and self-interest in peer to peer networks. *Proc. of the 2nd Intl. Workshop on Peer-to-Peer Systems*, pp. 139–148, Springer-Verlag, USA, February 2003.

[131] Y. Shu, B. C. Ooi, K-L. Tan, and A. Zhou. Supporting multi-dimensional range queries in peer-to-peer systems. *Proc. of the 5th Intl. Conf. on Peer-to-Peer Computing*, pp. 173–180, IEEE Computer Society Press, Germany, August 2005.

[132] D. Spence and T. Harris. XenoSearch: Distributed resource discovery in the XenoServer open platform. *Proc. of the 12th IEEE International Symp. on High Performance Distributed Computing*, pp. 216–225, IEEE Computer Society Press, USA, June 2003.

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

[134] D. Stutzbach, R. Rejaie, and S. Sen. Characterizing unstructured overlay topologies in modern P2P file-sharing systems. *Proc. of the 2005 Internet Measurement Conf.*, pp. 49–62, USENIX Association, USA, May 2005.

[135] C. Tang, Z. Xu, and S.Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. *Proc. of ACM SIGCOMM*, pp. 175–186, ACM Press, Germany, August 2003.

[136] Y. M. Teo, V. March, and X. Wang. A DHT-based grid resource indexing and discovery scheme. *Proc. of Singapore-MIT Alliance Annual Symp.*, Singapore, January 2005.

[137] R. Tian, Y. Xiong, Q. Zhang, B. Li, B. Y. Zhao, and X. Li. Hybrid overlay structure based on random walks. *Proc. of the 4th Intl. Workshop on Peer-to-Peer Systems*, pp. 152–162, Springer-Verlag, USA, February 2005.

[138] D. Tsoumakos and N. Roussopoulos. A comparison of peer-to-peer search methods. *Proc. of the Intl. Workshop on Web and Databases*, pp. 61–66, USA, June 2003.

[139] J. Xu. On the fundamental tradeoffs between routing table size and network diameter in peer-to-peer networks. *Proc. of INFOCOM*, pp. 2177–2187, IEEE Press, USA, March 2003.

[140] Z. Xu, R. Min, and Y. Hu. HIERAS: A DHT based hierarchical P2P routing algorithm. *Proc. of the 2003 Intl. Conf. on Parallel Processing*, pp. 187–194, IEEE Computer Society Press, Taiwan, October 2003.

[141] B. Yang and H. Garcia-Molina. Improving search in peer-to-peer networks. *Proc. of the 22nd IEEE Intl. Conf. On Distributed Computing Systems*, pp. 5–14, IEEE Computer Society Press, Austria, July 2002.

[142] B. Yang and H. Garcia-Molina. Designing a super-peer network. *Proc. of the 19th Intl. Conf. on Data Engineering*, pp. 49–61, IEEE Computer Society Press, India, March 2003.

[143] B. Y. Zhao, Y. Duan, L. Huang, A. D. Joseph, and J. Kubiatowicz. Brocade: Landmark routing on overlay networks. *Proc. of the 2nd Intl. Workshop on Peer-to-Peer Systems*, pp. 34–44, Springer-Verlag, USA, March 2002.

[144] B. Y. Zhao, J. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, Computer Science Department, UC Berkeley, April 2001.

[145] C. Zhu, Z. Liu, W. Zhang, W. Xiao, Z. Xu, and D. Yang. Decentralized grid resource discovery based on resource information community. *Journal of Grid Computing*, 2(3):261–277, September 2004.

[146] Y. Zhu, X. Yang, and Y. Hu. Making search efficient on Gnutella-like P2P systems. *Proc. of the 19th IEEE Intl. Parallel and Distributed Processing Symp.*, pp. 56a, IEEE Computer Society Press, USA, April 2005.