Content-Addressable Network for Distributed Simulations

Der Fakultät für Ingenieurwissenschaften der

Universität Duisburg-Essen

Abteilung Informatik und Angewandte Kognitionswissenschaft

zur Erlangung des akademischen Grades eines

Doktor der Ingenieurwissenschaften

genehmigte Dissertation

von

Zhongtao Li

aus

Shandong, VR China

Referent: Prof. Dr.-Ing. Torben Weis

Koreferent: Prof. Dr. Arno Wacker

Datum der mündlichen Prüfung: 27.10.2014

Abstract

The development of distributed systems, parallel computation technology, and Peer-to-Peer systems facilitates the realization of a distributed interactive world model. Thereby, we can implement a worldwide distributed simulation and virtual community, e.g., city traffic simulation and Massively Multiuser Virtual Environments (MMVE).

In this thesis, we present Content-Addressable Network for Simulations (CANS), which is based on CAN. Thus, it incorporates all the advantages of CAN, such as self-organization, scalability, and fault-tolerance. The peers in CANS carry out the simulation for the zone assigned to them, and the zones are allocated in such a way that there is as little communication between the peers as possible. We propose two approaches for reorganizing zone-assignments after peers churn. These approaches are based on the distributed tree structure and prefix code. In comparison to existing approaches, our proposed approaches are more efficient and reliable.

Since CANS is used to simulate “city traffic” and MMVE, it requires a low-dimensional key space, i.e., a two-dimensional or three-dimensional key space. Thus, we propose CAN tree routing and zone code routing, both of which adopt long links. CAN tree routing has a hierarchical design that is based on the CAN tree. Each peer equips two long links on average. Zone code routing is based on B*-tree. Each peer equips $\log_2 n$ long links and shares the load evenly. Both of these routing solutions achieve $O(\log n)$ routing hops on average.

Consequently, the existing CAN can be optimized to perform simulations efficiently and reliably.

Acknowledgment

I would like to take the opportunity to thank everybody who aided and fostered me. Without their aid, it would have been much more burdensome to complete this thesis.

First of all, I would like to thank Prof. Dr.-Ing. Torben Weis for giving me the opportunity to work on this thesis and for the freedom to explore my own ideas in the execution of my research. His inspiration and support made my time at academia a very pleasant one. Without his guidance and encouragement, I would not have succeeded in finishing this thesis.

My gratitude is extended to my colleagues and friends in Duisburg-Essen University who accompanied me throughout the long and winding journey until the completion of this work. Special thanks Bernd Holzke and Marianne Appelt for organizing everything we needed.

Last but not least, I am deeply grateful to my family and my friends. They gave me the loving care and emotional support, shared their inspiring experiences and were always there for me.

Zhongtao Li

Duisburg, August 2013

Content

List of Figures

List of Tables

Chapter 1

Introduction

Advances in distributed systems and parallel computation technology enable us to realize distributed interactive world models. Using distributed world models, we can implement worldwide distributed simulations and virtual communities, e.g., city traffic simulation and Massively Multiuser Virtual Environments (MMVE) [1]. The distributed world model uses a Peer-to-Peer architecture to eliminate the need for fixed, expensive hosting infrastructure [2]. In the Peer-to-Peer architecture, any user who wants to join has to provide his/her share of bandwidth, CPU, and disk space [3]. It is an alternative to the older central server or server cluster. A computer cluster consists of a set of stable and efficient connected computers that work together. However, in order to establish large-scale models using computers distributed worldwide, an appropriate infrastructure is needed.

Users expect these virtual environments to react to their actions instantaneously. Thus, state updates must be propagated with very little delay. Recently, the public network infrastructure has rapidly developed to the point where optical fiber communication, in particular, is widely available. Public networks are increasingly providing more high quality services. As a result, worldwide distributed interactive world models are able to fulfill the highest requirements with respect to scalability and consistency.

In contrast to server clusters, peers are highly dynamic in the distributed interactive world model, especially when the peers are personal computers. Each peer may join, leave, or crash at any time. Thus, an appropriate Peer-to-Peer protocol is needed to implement self-organizing peers. Each peer is responsible for partial computation; all peers comprise complete functionality. When a peer crashes, other peers must extend

their responsibility to take over the functionality of the crashed peer. In doing so, the distributed interactive world model provides functionality that is intact to all users.

We first address the fundamental problems of the distributed world model. We then show how to split and merge the virtual world to maintain efficient simulation, and explain the routing mechanisms that allow efficient routing in low-dimensional virtual space, such as two-dimensional and three-dimensional space.

1.1 The CANS Idea

Our underlying idea is a Peer-to-Peer based distributed world model. In this model, in contrast to the classic heavyweight server-based infrastructure, computing overhead is distributed over all participating peers using Peer-to-Peer technology. We assume that each peer uses the same Peer-to-Peer protocol, such as Gnutella, Chord [4], Pasty [5] or Content-Addressable Network (CAN) [6, 7], to combine and cooperate. Structured approaches to Peer-to-Peer architectures have been proposed in order to improve efficiency, scalability, and fault-tolerance. Thus, a structured Peer-to-Peer protocol is an ideal infrastructure for a distributed interactive world model. The approaches are based on similar designs, while their search and management strategies differ. Ring-based approaches such as Pastry, and Chord all use similar search algorithms such as binary ordered B^*-tree. CAN is based on Geometry [8]. We chose CAN as the basis for our Content-Addressable Network for Distributed Simulation (CANS). Since CANS is based on CAN, it is designed to adapt to a changing number of peers and it can scale well, i.e., simulation speed improves as new peers join CANS.

1.1.1 Managing a Content-Addressable Network for Distributed Simulations

CANS is an improvement over conventional CAN for simulations. CANS is designed to handle simulations such as city traffic and MMVE. For example, Figure 1.1 shows how CANS with five peers simulates city traffic. Zones are then assigned to peers

using one-to-one mapping. In this small example, CANS divides the entire area into five zones. To a high degree, peers can run the simulation of their zone independently. Peers only need to communicate to synchronize their simulation efforts, or hand over players or cars crossing zone boundaries. In order to improve efficiency, we must reduce communication between peers.

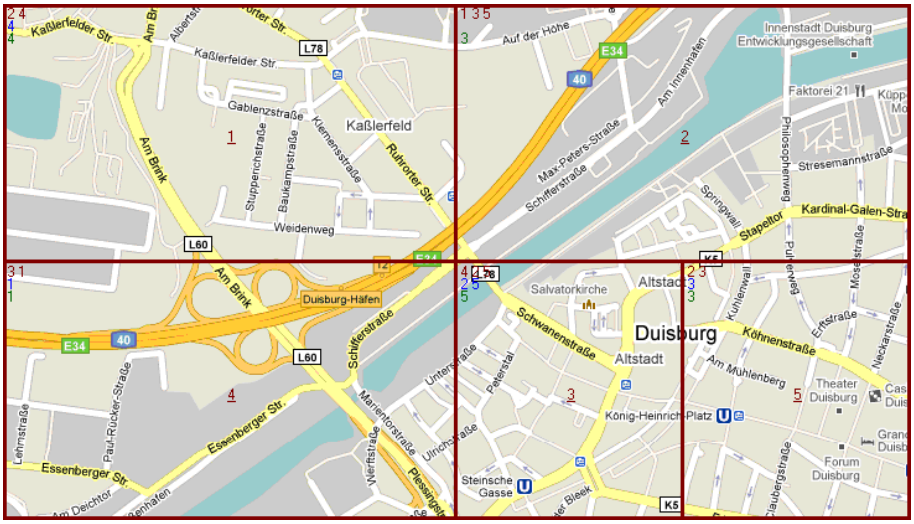
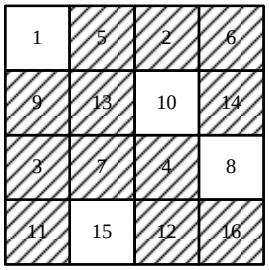


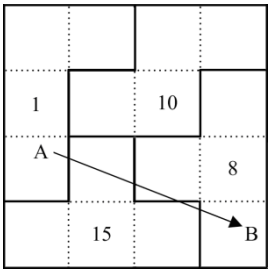
Figure 1.1 City traffic simulation

When peers leave or crash, we must ensure that their zones are handled by another existing peer. After taking over the zone of another peer, a peer may have to handle multiple zones or a polygon zone. In CAN-like file sharing, peers hardly ever communicate with each other, and so these polygon zones are acceptable. However, constant communication in CANS generates extra load; and the extra load will never disappear until these zones are merged. For example, CANS has 16 peers, and every peer handles only one zone. When some peers leave (the departing peers are shaded in Figure 1.2(a)), their neighbors take over their zones. If zones are arbitrarily merged with the zones of departed peers, the result may be a concave polygon (see Figure 1.2(c)). Otherwise, every peer handles multiple zones (see Figure 1.2(b)). Concave polygon and multiple zones increase the communication between peers and generate extra cost. In city traffic simulations, when a car drives across the border, peers must communicate with each other. If a car drives from location A to location B, it crosses

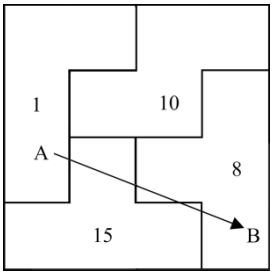
the boundaries four times in a polygon zone or multiple zones. If zones are convex (see Figure 1.2(d)), the car crosses the boundary only once.



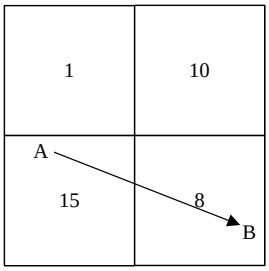
(a) Departing peers shaded



(b) Peers handle multiple zones



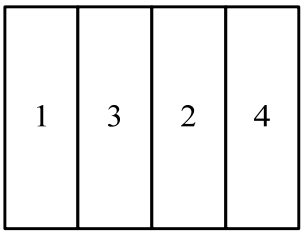
(c) Peers handle concave polygon zones



(d) Peers handle convex zones

Figure 1.2 CANS with 16 peers

In another case, cars will often cross the boundaries because of slim zones (see Figure 1.3). The aforementioned two problems are termed the “concave and slim” problems.



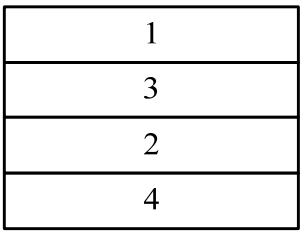


Figure 1.3 Slim zones

We try to keep CANS running with a simple structure and high efficiency; therefore, our peers must have neither multiple zones nor a polygon zone. We enforce the rule that every peer handles exactly one hyper-rectangular zone, whose edges are in proportion.

1.1.2 Routing Mechanisms for Content-Addressable Network for Distributed Simulations

The original routing mechanism in CAN has the lowest efficiency among structured Peer-to-Peer routing mechanism. The routing hops of Chord are $O(\log n)$ on average for a Chord circle with *n* participating peers. In Pastry with *n* peers, the target is reached in $\log_{2^b}(n)$ hops where b is typically chosen to be 4. CAN can only forward messages to immediate neighbors which are closer to the destination coordinates (greedy routing). Hence, greedy routing is not very efficient, particularly in large-scale dynamic CAN. Because CAN has a *d*-dimensional key space, routing efficiency and *d* are correlated. CAN routing complexity is $O(d \cdot n^{1/d})$ in a *d*-dimensional key space.

We know the average routing path length, i.e., the number of peers traversed during routing, from the CAN simulator. Figure 1.4 [6] illustrates the average routing path length in each case for dimensions two to five. The results indicate that more dimensions result in lower average routing hops. In order to reduce the routing hops in CAN, researchers have proposed increasing the number of immediate neighbors per peer by enhancing dimensions. In higher dimensional CAN, each peer has more immediate neighbors..

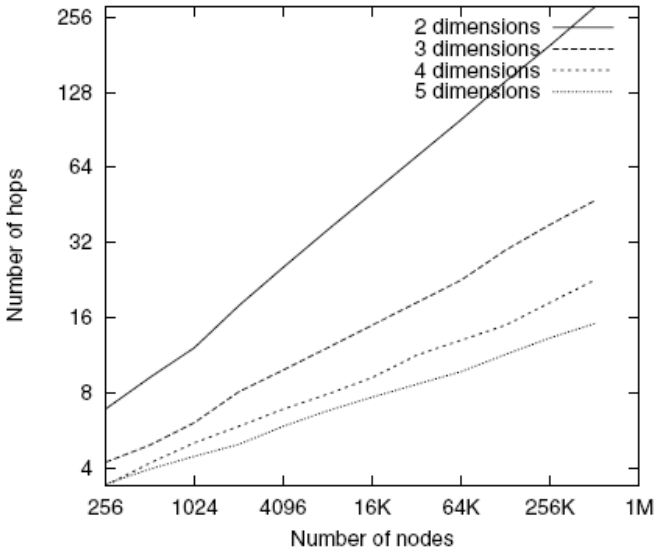


Figure 1.4 Effect of dimensions on path length

People also proposed long link solution. The routing is not limited between immediate neighbors (Figure 1.5). The message could forward to a further distant neighbor via long link. Then long link routing reduces routing hops.

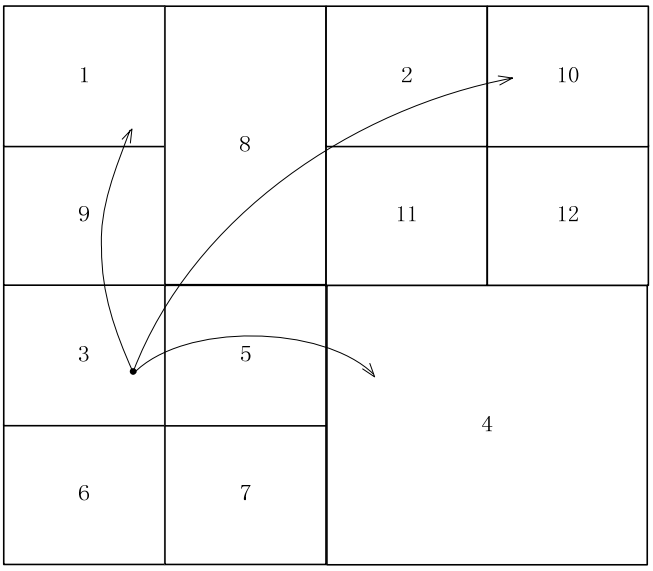


Figure 1.5 Long links and distant neighbors

Higher dimensions solution and long link solution need to append more links to reduce routing hops. Because our CANS is utilized to simulate “city traffic” and MMVE, which are two-dimensional or three-dimensional spaces, we need efficient

routing solutions for low dimensions. Therefore, the long link solution is suitable for CANS.

1.2 Contributions

In this thesis we present two algorithms to manage key space reorganization after peer departure. Using these algorithms, CAN is adapted for distributed simulation systems from distributed file-sharing system[9]. We also propose two routing solutions that perform efficiently in low dimensions

More specifically, the main contributions of this thesis are as follows:

- We derive and analyze the requirements for setting up a Peer-to-Peer distributed simulation system and uncover the “concave and slim” problem that limits simulation efficiency.
- We derive and analyze the requirements for CANS routing. Routing improvement schemes that are based on the enhancing of dimensions are unsuitable for CANS. Hence, we need efficient routing solutions in low dimensions.
- We propose the CAN tree to solve the “concave and slim” problem. Using the distributed CAN tree, our system maintains low overhead and automatically adapts to network changes.
- The zone code is another solution to the “concave and slim” problem. After peers churn, CAN tree adapts to the changes in CAN via peer communication. However, the zone code solution does not need the update communication. It further reduces the overhead of CANS.
- CAN tree routing is designed to efficiently send messages in low-dimension CAN. Therefore, it is an appropriate solution for CANS. Since each peer maintains only two extra long links, the extra overhead is very cheap.

- Since CAN tree routing is based on tree infrastructure, the peers have unfair routing overhead. However, zone code routing overcomes this drawback, and achieves $O(\log n)$ on average.

We complement these contributions with a detailed performance evaluation of our algorithms.

1.3 Thesis Organization

The central focus of this thesis lies on the design and evaluation of a CAN for distributed simulation and routing. This thesis has demonstrated that existing Peer-to-Peer technologies can be optimized to interesting application domains which are not at all related to file sharing. Accordingly, the remainder of the thesis is structured as follows:

Chapter 2: State-of-the-Art and Related Work

This chapter introduces the technologies used in this thesis and related works. We thereby give an overview of existing Peer-to-Peer technologies and different solutions from other researchers.

Chapter 3: Using CAN Tree to Manage a CANS

In this chapter we present an efficient implementation, the so-called CAN tree. In order to solve “concave and slim” problem, we need CAN splitting history. The idea of this implementation is to use a distributed tree structure to record CAN splitting history. When peer leaves or crashes, we recover system using CAN tree.

Chapter 4: Using Zone Code to Lookup Mergeable-zones

After showing how CAN tree records the splitting history and searches mergeable-zones efficiently, we would like to decrease communication cause of peers churn. We present zone code to do the same work with the CAN tree, with no

communication needed among the peers. Consequently, the zone code scheme reduces communication overhead.

Chapter 5: CAN Tree Routing

This chapter encourages the need for routing in a CANS in chapter 3 and 4. In order to reduce the routing latency in CAN, the original CAN proposed to increase the number of immediate neighbors per peer by enhancing the dimensions. However, our CANS is utilized to simulate “city traffic” and MMVE. They require a two-dimensional or three-dimensional space. Hence, we present an efficient tree infrastructure routing solution to overcome the weakness of greedy routing.

Chapter 6: Zone Code Routing

CAN tree routing in chapter 5 is a tree infrastructure. The routing performance can be boosted from $O\left(n^{1/d}\right)$ to $O(\log n)$ by equipping each peer with two long links on average. However, the tree infrastructure causes unfair overhead. Hence, we map d-dimensional zones onto a one-dimensional zone code space and routing in zone code space. Zone code routing achieved $O(\log n)$ routing performance with $O(\log_2 n)$ routing state per peer.

Chapter 7: Conclusion

This chapter closes this thesis with a summary of the most important findings. Furthermore, it gives an outlook on interesting future work areas.

Chapter 2

State-of-the-Art and Related Work

Peer-to-Peer (P2P) has demonstrated that it is a powerful paradigm for utilizing distributed resources and performance critical functions in a decentralized manner [10].

2.1 Peer-to-Peer Systems

Peer-to-peer systems spread computing and data storage to peers. They offer a large variety of benefits such as massive scalability and reliability, better resource utilization, and fault-tolerance. However, it is difficult to find a generally accepted definition for Peer-to-Peer systems. Oram gives a basic definition of the term "Peer-to-Peer" that is further refined in [8, 11]:

"(A Peer-to-Peer system is) a self-organizing system of equal, autonomous entities (peers) (which) aims for the shared usage of distributed resources in a networked environment avoiding central services" [8].

Scalability

In a centralized system, the number of users is limited by system resources such as bandwidth, storage capacity, and the processing power of certain applications. Usually peers have many spare resources that can be contributed to the P2P system at no cost. Hence each individual runs underutilized most of the time, if a sufficient number of peers participate in the system. Vast resources are spread over all users; therefore, a Peer-to-Peer system can scale several orders of magnitude without loss of efficiency [8]. In contrast, centralized systems are less cost-efficient [12].

Reliability

Reliability is critical for availability. Systems must survive crashing peers and network failures. If some peer craches, this should not affect the others. However, the system may experience partially failures. The surviving peers could recover the system by means of replicating all data across multiple peers.Network failure could result in isolated peers. Because every peer runs independently, we must ensure data consistency after network recovery.

Self-organization

Self-organizing systems must solve all problems by themselves. Even if some peers crashed in a system, the system must survive and regain full functionality. A Peer-to-Peer system assigns areas of the key space to individual peers in such a way that the areas assigned to peers do not overlap and there are no gaps in the key space. In order to realize self-organization, a peer-to-peer system needs a protocol to handle peer churn. The key space splitting needs to be robust when peers join or leave the system [8].

2.1.1 Centralized Peer-to-Peer Systems

Some systems rely on locating a central dedicated server, which stores the locations of all data items, and assumes responsibility for mapping “keys” onto “values”[13]. After retrieval of the location of a data item via the dedicated server, peers directly access and exchange the shared data item. Such systems are also termed centralized Peer-to-Peer systems, e.g., Napster (see Figure 2.1)[14, 15].

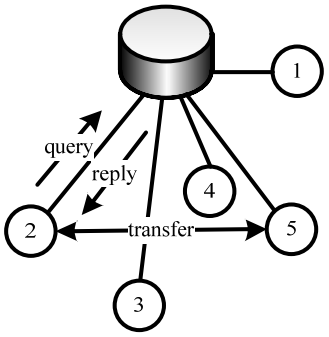


Figure 2.1 Dedicated server

Napster[16] is one of the most popular typical representatives. It was originally established as a pioneering Peer-to-Peer file sharing Internet service. All users were able to not only download files, but also share files with other participating users via a central server. The server maintained the IP addresses of participating peers without storing the files. It collected information about files being shared and offered an index of all files available for sharing [12]. When a peer desired to download, it looked up a potential providers list on the server in a query-response fashion. In addition, the peer directly downloaded the file from one of the remote providers without the server.

Napster is invariably susceptible to single points of failure. The central server assumed responsibility for the lookup/index. Without it, no files could be exchanged. Regardless of server fault, the system offered less bandwidth for each peer as more peers joined. The server posed a bottleneck problem [12].

2.1.2 Unstructured Peer-to-Peer Systems

Unstructured Peer-to-Peer systems use a flooding technique whose “lookup” queries are sent to all participating peers in the system. The peer, which covers the corresponding data item, replies and transfers the data directly (see Figure 2.2)[17]. Thus, these systems do not rely on any central server (except a bootstrap server to ease joining network). They are decentralized systems and are termed pure Peer-to-Peer systems, e.g., Gnutella 0.4[18] and Freenet[19].

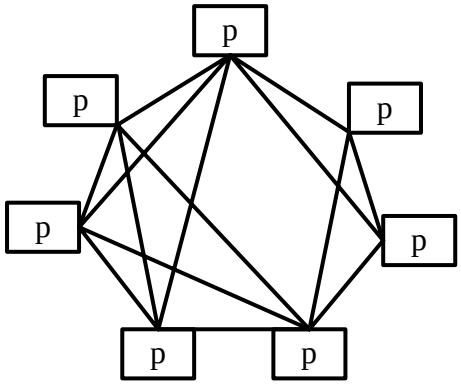


Figure 2.2 Pure Peer-to-Peer system

Gnutella is one of the most popular typical representatives of pure Peer-to-Peer systems (see Figure 2.3). In order to detect other active peers, all peers broadcast ping messages and reply with pong messages to echo received ping message. The use flooding queries to look up a desired file. Every peer sends any incoming query that had not been received before to all neighbors except the peer from which the query originated [8]. If a query had been received, peer will never forward it to any other peer. In this way, queries are further flooded peer by peer, until the Time-to-Live (TTL) value reaches zero [12, 20].

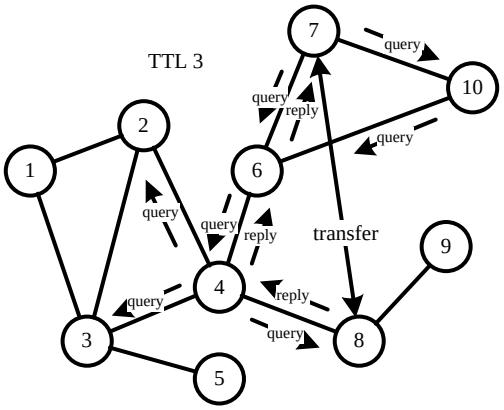


Figure 2.3 Gnutella 0.4

Flooding queries generates potentially huge amounts of network traffic. Large network traffic causes packets to collide in dense networks. The Time-to-Live (TTL) mechanism improved the pure Peer-to-Peer system, however did not solve all the

problems it experienced. Furthermore, TTL reduced the probability of retrieving the desired file [12].

In order to overcome the drawbacks experienced by pure Peer-to-Peer systems (system without dedicated server), a hierarchy system is proposed to avoid unnecessary traffic. Super-peers (see Figure 2.4), which store the content available at the connected peers together with their IP address [8], were proposed. Super-peers are able to immediately answer requests instead of the respective peers, and fewer hops are required in the search process. It is a kind of hybrid Peer-to-Peer system. It achieves a balance between the perfect index of available files (centralized system) and evenly distributing load on all peers (pure Peer-to-Peer system). Hybrid Peer-to-Peer systems have a hierarchical structure. Peers that have a more powerful processor and more bandwidth are termed super-peers. Peers connect only to super-peers instead of each other, and super-peers store information about the peers that are connected to them. They perform as centralized servers in interconnection and maintain connections with other super-peers. Each peer sends its query to its super-peer when searching for a desired file. The super-peer forwards the query to other super-peers if it is not in charge of the peer with the desired file [12]. For example, Gnutella 0.6 [21] and JXTA [22] are hybrid Peer-to-Peer systems.

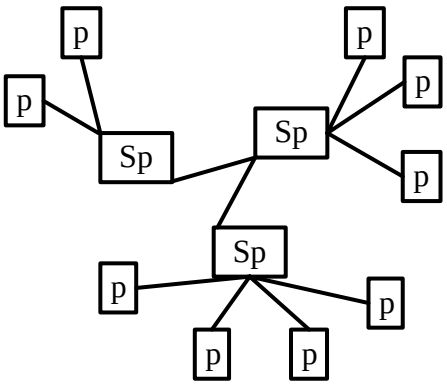


Figure 2.4 Centralized super-peers

Figure 2.5 illustrates Gnutella 0.6 relaying a Query-Hit message and a file being directly transferred between peers.

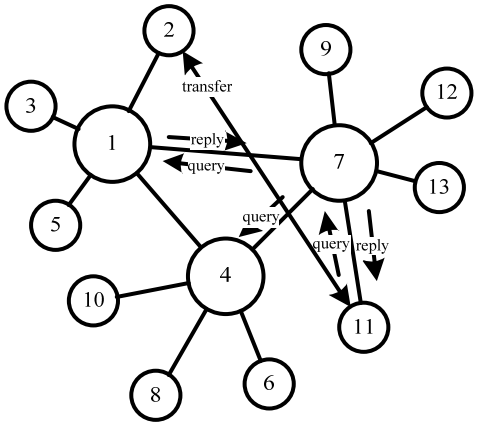


Figure 2.5 Gnutella 0.6

In some hybrid Peer-to-Peer systems, regular peer and super-peer offer different functionality. We differentiate peers in regard to their role. For instance in FastTrack[23] (Kazaa's search engine) there are regular peers and super-peers. Those super-peers offer different searching functionality than the regular peers do[12].

The above schemes do not follow any specific structure, and the content stored on a peer is not related to its peer ID. Hence, they are generally termed “unstructured Peer-to-Peer” systems.

2.1.3 Structured Peer-to-Peer Systems

Structured Peer-to-Peer systems were proposed in an effort to improve efficiency, scalability, and fault-tolerance[24]. They manage data via Distributed Hash Tables (DHTs) and adopt a routing scheme that allows any user to efficiently look up the peer covering a specific data item. Table 2.1 lists the characteristics of the approaches presented in terms of state per peer (the number of neighbors), communication overhead. Distributed Hash Tables cope best with accurate queries; however, for fuzzy or semantic queries, unstructured Peer-to-Peer systems are still the best option.

System	State per Peer	Communication Overhead	Fuzzy Queries
Central Server	$O(n)$	$O(1)$	√
Flooding Search	$O(1)$	$\geq O(n+E)$	√
Distributed Hash Table	$O(\log n)$	$O(\log n)$	×

n is the number of peers. E is the number of connections.

Table 2.1 Characteristics of Peer-to-Peer systems [8]

DHTs have a lot of variants such as Chord, Pastry, CAN, Tapestry[25], p-Grid[26], Viceroy[27], Kademlia[28], or the Continuous-Discrete Approach[29-31]. All these Peer-to-Peer systems implement the similar service (DHTs). However, they differ in the topology of the overlay network that has an impact on the efficiency of the different operations in a DHT.

Chord

Chord [4] operates on a one-dimensional key space (*l*-bit identifiers, i.e., integers in the range $[0,2^l-1]$), and creates a circular structure (see Figure 2.6). Each data item and peer is assigned an identifier in key space. Each peer stores key-value pairs. The key-space is divided among the peers in such a way that each peers is responsible for the keys which are equal or less than its peer ID and large than the peer id of the predecessor. All distributed key-value pairs form the DHT.

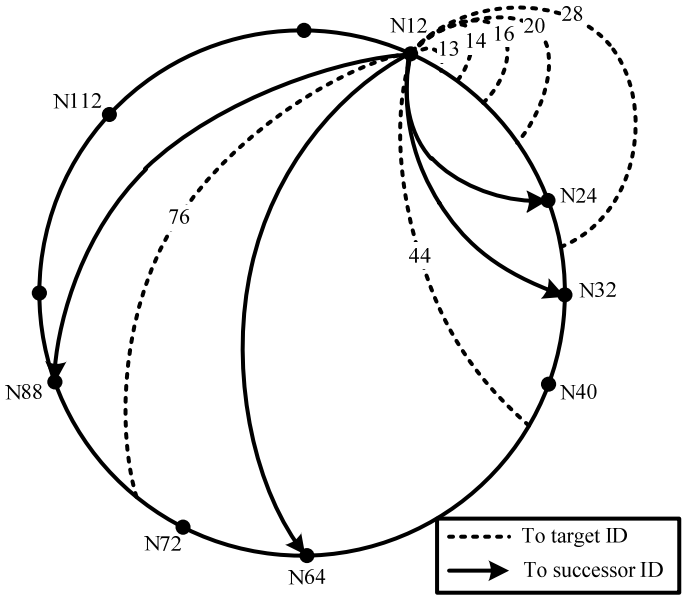


Figure 2.6 Chord

To facilitate efficient lookup, each peer needs to build long links to its successor peers on the circular key space. When a peer looks up a key, it sends the query to its successor peer. If the successor peer determines that the key is not located between itself and its predecessor, it forwards the query to its successor peer. Otherwise, the key must be stored by the successor peer. Therefore, the successor peer replies and transfers the desired file. The linear lookup on the circular key space is inefficient. Chord utilizes a finger table (see Figure 2.7) to enhance lookup speed.

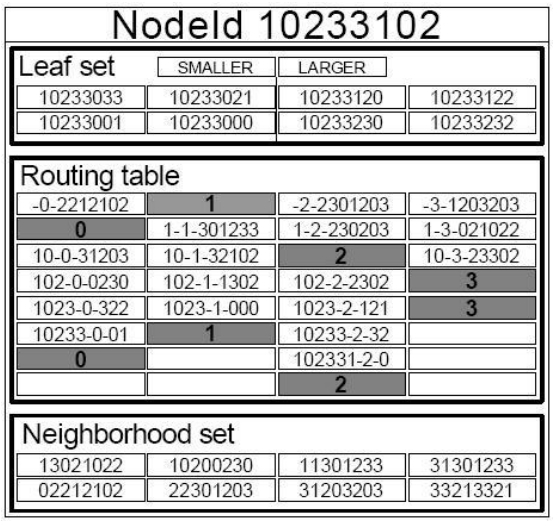
Finger Table Node 12		
No.	Taget ID	Successor
1	$12+2^0=13$	N24
2	$12+2^1=14$	N24
3	$12+2^2=16$	N24
4	$12+2^3=20$	N24
5	$12+2^4=28$	N32
6	$12+2^5=44$	N64
7	$12+2^6=76$	N88

Figure 2.7 Finger table of Chord

Due to the fact that the circular key space is *l*-bit identifiers, each peer maintains a finger table with *l* entries. For peer *n*, the entry of row *i* is a successor peer that is the first successor of $(n+2^{i-1}) \bmod 2^l$, e.g., Figure 2.7 illustrates the finger table of peer 12 in 7-bit Chord [4]. By means of the finger table, each hop covers at least half the clockwise distance between the current peer and the target peer. Hence, routing complexity is $O(\log n)$ with *n* participating peers.

Pastry

Pastry was proposed by Rowstron and Druschel in [5, 32]. It is similar to Chord. Pastry operates a circular ID-space that ranges from 0 to $2^l - 1$, and uses key-value pairs to map data items to ID-space. Pastry concentrates not only on reducing routing hops, but also on geographical location.



NodeId 10233102

Leaf set	SMALLER	LARGER	
10233033	10233021	10233120	10233122
10233001	10233000	10233230	10233232

Routing table

-0-2212102	1	-2-2301203	-3-1203203
0	1-1-301233	1-2-230203	1-3-021022
10-0-31203	10-1-32102	2	10-3-23302
102-0-0230	102-1-1302	102-2-2302	3
1023-0-322	1023-1-000	1023-2-121	3
10233-0-01	1	10233-2-32	
0		102331-2-0	
		2	

Neighborhood set

13021022	10200230	11301233	31301233
02212102	22301203	31203203	33213321

Figure 2.8 State of a Pastry peer with ID 10233102, *b* = 2, and *l* = 8 (base 4) [5]

Pastry routing information comprises routing table, leaf set, and neighborhood set (see Figure 2.8). The identifiers of pastry are strings of digits to the base 2^b where b is typically chosen to be 4. The routing table of peer n is made up of $\frac{l}{b}$ rows with $2^b - 1$ entries per row, and all entries in row *i* map to peers whose identifiers share *i*-digit prefix with peer *n* but differ in digit *i*+1 [8]. Pastry enhances lookup efficiency

via routing table by storing prefix-ID peers. The leaf set L is formed by a set of peers, which are the numerically closest peer IDs to the current peer ID, and uniformly distribute on both sides, i.e., clockwise and counter-clockwise. The neighborhood set M is comprised of $|M|$ peers that are geographically near the current peer [33]. "Geographically near" is proximity metric and measures the amount of IP hops or the ping latency [12]. They directly reflect the distance between peers. The neighborhood set does not participate in routing but in maintaining network locality in the routing information.

Routing in pastry consists of two steps. In the first step, a peer checks whether the key falls into the range of its leaf set peers. If this is the case, it implies that the peer can directly send queries to the peer numerically closet to the key. Thus, the routing process is finished. Otherwise, the key is not covered by its leaf set. In this case, the query needs to be sent over a longer distance peer. The current peer therefore looks up in its routing table a peer that shares longer common prefix with the key than itself. If the suitable peer is not reachable, the current peer sends the query to a peer that shares at least the same prefix with itself, and whose ID is however numerically nearer to the key [33].

If the key falls into a leaf set, it always needs one hop to deliver the query to the target peer. If the key is forwarded via the routing table, the number of peers with longer prefixes is reduced by the factor 2^b in each hop. Thus, the routing needs $\lceil \log_{2^b} n \rceil$ hops. Given that the routing table might not offer a peer with a longer prefix, this case leads only to one additional routing hop. Hence, the expected number of pastry routing steps is $\lceil \log_{2^b} n \rceil$ [12].

2.2 Bootstrapping

To join an overlay network, a new peer must discover at least one of the participating peers as entry because the new peer does not have a global view of the overlay

network. This means that it does not even know whether the overlay network exists[34]. The search for an overlay network is a critical problem that is termed bootstrapping[35].

Peers may join it at any time, thereby becoming a part of the overlay network and taking responsibility. Analogously, any peer might leave the overlay network at any time without announcement. Therefore, the size of a Peer-to-Peer network dynamically varies from zero to all potential peers in the network [36]. Furthermore, the process of bootstrapping has to be performed via minimal bandwidth consumption. In addition, the following four properties should be simultaneously achieved [8]:

1. **Availability**: Availability is one of the most important properties of bootstrapping. A probabilistic approach is not sufficient. Thus, the bootstrapping mechanism must perform well at any time, i.e., whenever a new peer needs network entry, it can retrieve one. Additionally, the system should be realized by decentralized infrastructure to avoid a single point of failure [36].
2. **Self-organization**: During the bootstrapping process, it must perform automatically and without any manual interaction [36].
3. **Efficiency**: In order for the bootstrapping mechanism to perform efficiently, the mechanism accepts a new peer within a reasonable amount of time and minimal bandwidth consumption [12].
4. **Scalability:** The system has to make sure that system overhead does not increase as an increasing number of peers join [12].

Bootstrapping is distinguished between two classes: peer-based approaches and mediator-based approaches [37].

Peer-based Approaches

Peer-based approaches try to detect peers in the overlay by contacting other peers directly. Peer-cache is one of the most popular typical representatives. A peer-cache contains a list of previously known peers that are potentially participating peers[38].

By trying to contact peers in its peer-cache, a peer possibly discovers existing peers in the overlay. Any peer that replies can be used as an entry point into the overlay. This approach is straightforward and efficient. However, it cannot guarantee one hundred percent success, e.g., all peers in its cache may have left the overlay.

Mediator-based Approaches

Mediator-based approaches use a well-known entry point as the mediator to provide assistance. In contrast to peer-based approaches, a peer (or some peers) maintains a participating peers list and determines which peer in the list should be offered to the new peer as the entry point. Mediator-based approaches easily balance load between peers. However, updating the available peers list consumes significant bandwidth.

2.3 Two-dimensional Peer-to-Peer Systems

The decentralized Peer-to-Peer systems have been proposed for storage ensuring reliability, such as Chord [4] or Pastry [5]. Recent research has shown that one can use such networks to build two-dimensional Peer-to-Peer system[39]. In order to assign a Chord or Pastry peer corresponding to the two-dimensional space, the space is divided into equally-sized zones (see Figure 2.9). Each zone may contain at most one peer. The more zones, the more peers can be supported[40]. After a peer has booted and chosen a location[41] in the space, it could join the Chord or Pastry ring. Since Georg Cantor demonstrated that any two finite-dimensional smooth manifolds regardless their dimensions have the same cardinality[12], Peer-to-Peer system could map from two-dimensional space into one-dimensional DHT space depending on a suitable space-filling curve[42] solution[43, 44]. For example, Mirko Knoll proposed Geostry[12] that is a Peer-to-Peer System for location-based[45, 46] Information.

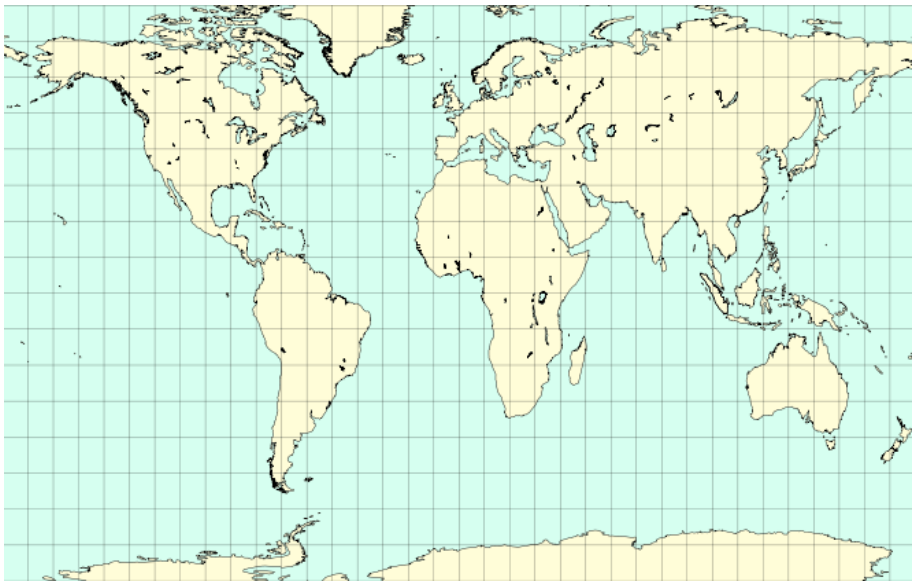


Figure 2.9 Divide space into equally-sized zones[47]

There are some solutions to implement space-filling curves. The simplest one is s-shaped that map an index curve onto an area is to superimpose the curve (see Figure 2.10)[48]. Since the geographically close peers may have a large discrepancy in their IDs, s-shaped curve is not very promising. For example, the first peer of the first two rows in Figure 2.10 are geographically close, however their IDs are far apart[49].

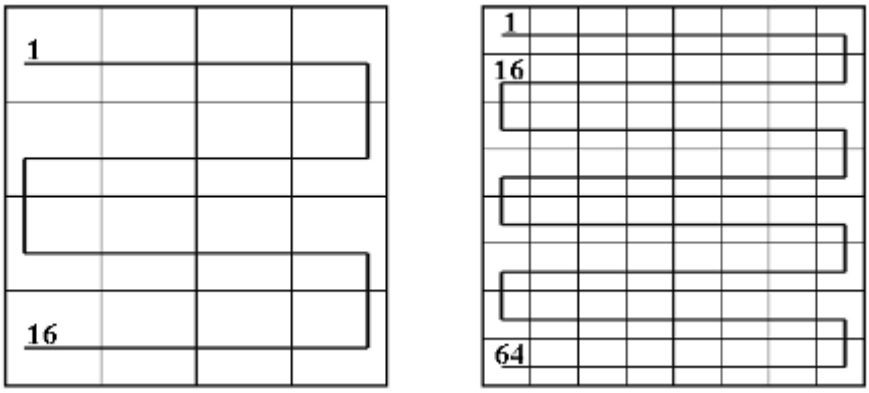


Figure 2.10 S-shaped space-filling curves[49]

Lebesgue Space-Filling Curve

Peano presented the space-filling curve that depends on further partitioning. On each partitioning step, each zone is divided into nine equal-sized sub-zones. And the curve

follows the initial mapping (see Figure 2.11). The distances between two adjacent zones on the curve is homogeneous[49].

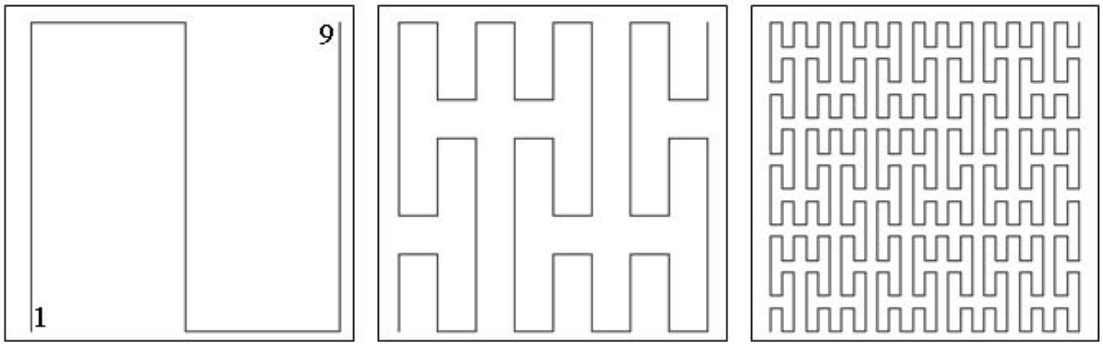


Figure 2.11 Peano curve[49]

Hilbert Space-Filling Curve

David Hilbert[50] proposed another curve which has better geometric locality properties in the worst case[51]. Hilbert space-filling curve starts with the basic "u"-form (see Figure 2.12)[52]. The order-two curve comprises four shrunken copies that are placed on the grid. While the position of the upper two curves matches their final orientation, the lower curves have to be rotated according to their position on the unit square (see Figure 2.12). The ends of curves which are facing each other are connected to form a continuous curve. In order to generate further-orders curve, the previous procedure is applied recursively[49].

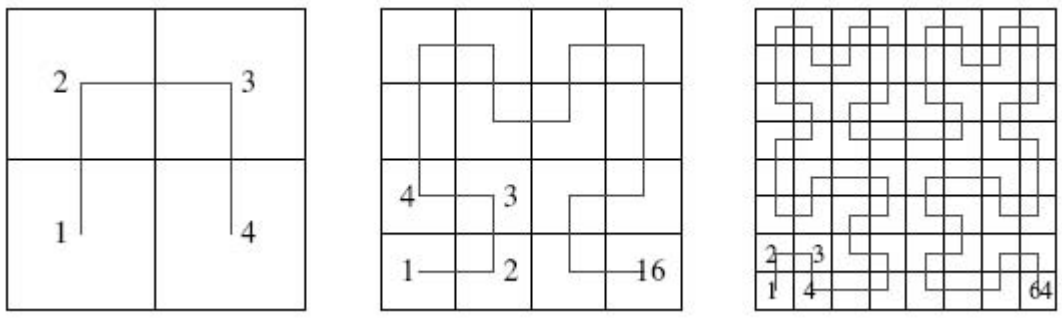


Figure 2.12 Hilbert Curve[49]

Such networks heavily rely on the Distributed Hash Tables (DHT). Since hashing destroys the locality of data[53], they hardly support efficient range queries[54].

However, our distributed simulation or MMVEs usually need retrieving all objects which are in a certain area[55]. Thus, we need two-dimensional Peer-to-Peer networks that use topologies defined in geometric space[6, 8], such as VoroStore [55] or CAN. They allow efficient range queries. VoroStore is based on the Voronoi diagram (see Figure 2.13) that is a special decomposition of key space. It is a complete solution for convex polygon zone partitioning depending on the locality of peers. Using replication, VoroStore can offer availability and ensure integrity of data.

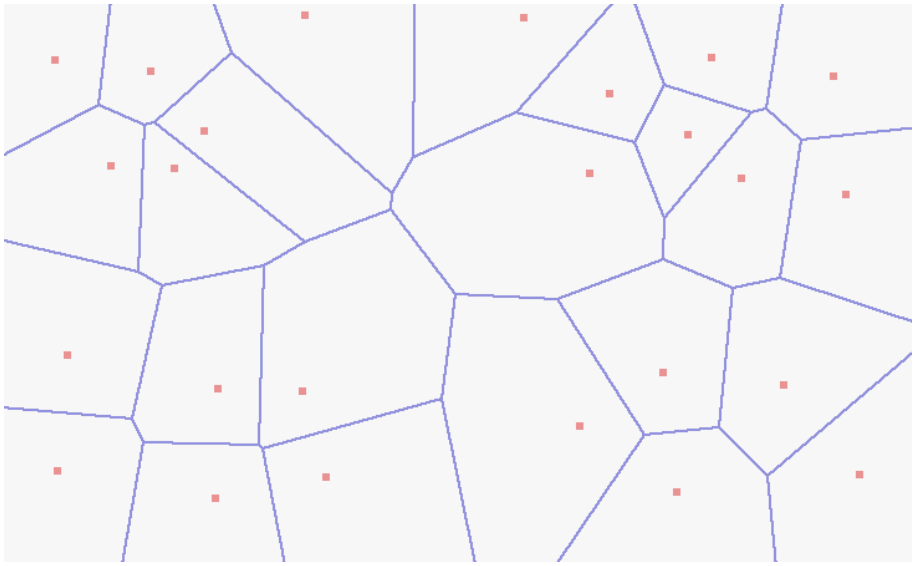


Figure 2.13 VoroStore: two-dimensional Peer-to-Peer network

Dominic Heutelbeck presented a distributed data structure for dynamic geometrical objects in [56]. It provided an abstract data structure called distributed space partitioning tree (DSPT). A DSPT is a general use distributed data structure, similar to distributed hash tables (DHTs), that allows publishing, updating of, and searching for geometrical objects (RectNet[57] is an implementation of DSPT, see Figure 2.14.). However, DSPTs allow the keys of the objects and queries to have a spatial extension with arbitrary boundaries.[57]

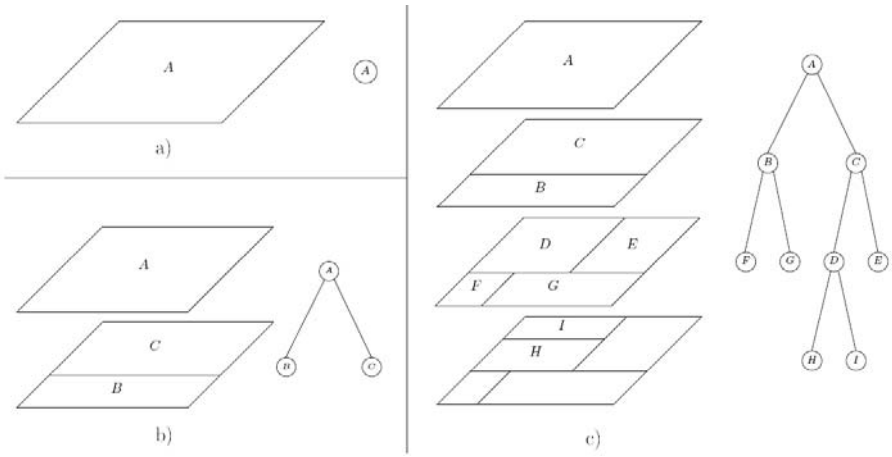


Figure 2.14 The network topology of RectNet[56]

When peer *B* distributes a message to all peers of zones intersecting area *K*, it reached that intersects with the target area *K*. Starting at *B*, each peer of a zone intersecting *K* forwards the message to all neighbors also intersecting *K*, except the neighbor from which it received the message itself. In addition, each peer caches the geographical messages it already distributed to its neighbors and does not send the same message to its neighbors twice. Since *K* is a connected subset of the context space, all peers intersecting *K* receive the message. [57]

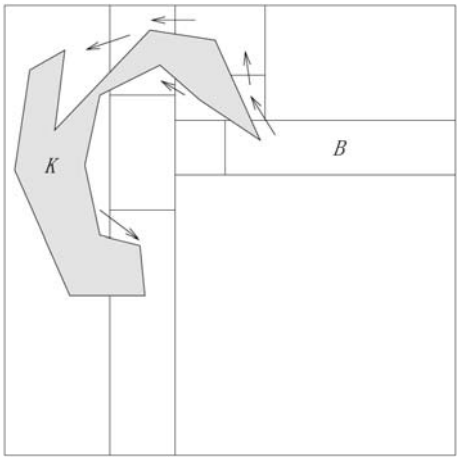


Figure 2.15 Flooding in concave area[57]

CAN uses a similar flooding scheme, described in [7], to provide application level multicasting. The directed flooding scheme described there reduces the number of

duplicated messages generated by the flooding. This approach can be used for peers that only contain convex zones. It is not suitable for concave zones[57]

However, DSPT allows the concave zones that generate extra communication overhead (see “concave and slim” problem in section 3.2). Hence, our system is based on CAN.

2.4 CAN

Content-Addressable Networks (CAN) is a well-known representative of Distributed Hash Tables (DHTs). CAN is used in large scale storage management systems such as OceanStore[9, 58] , Farsite[59] , and Publius[60]. Our Content-Addressable Network for Simulations (CANS) is derived from CAN. In this section, we describe in detail the functionality of CAN.

In 2001, Ratnasamy et al. [7, 61] proposed a novel DHT, Content-Addressable Network, which has a distributed, decentralized Peer-to-Peer infrastructure and provides DHT functionality on an Internet-like scale. CAN is scalable, fault-tolerant, and self-organizing.

2.4.1 Virtual Space

The design of CAN is based on a virtual *d*-dimensional Cartesian coordinate space (see Figure 2.16). The *d*-dimensional space is used to store key-value pairs. A key *k* is definitely mapped onto a point *p*, which is in the key space. As typical key-value pairs in DHTs, CAN keys are derived from the value by applying its hash function [8]. Points in the virtual space are identified with coordinates. Therefore, the peer, whose zone owns the point *k*, exclusively stores the corresponding (*k*, *v*). In order to retrieve the value *v* corresponding to key *k*, the requesting peer lookups in CAN. If the point *p* is located at the requesting peer, it retrieves the corresponding value *v* immediately. If this is not the case, the query is forwarded to the peer whose zone covers point *p* via the CAN infrastructure[6].

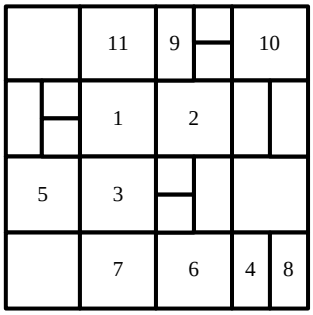


Figure 2.16 Two-dimensional Cartesian coordinate space

2.4.2 Peer

CAN is a highly distributed system, which built from thousands or even millions of typically non-dedicated peers through the Internet that might flexibly join or leave the system at any time [8]. Each peer performs a part of the functionality.

Zone

All peers in a CAN Peer-to-Peer overly network dynamically divide the entire key space into a number of non-overlapping hyper-rectangular zones. Each peer is assigned at least one distinct zone within the key space. When a new peer joins CAN, it is allocated its own portion of the key space. This is done by an existing peer splitting its allocated zone in half, retaining half and handing the other half to the new peer [6]. When sharing half a zone, the split peer splits its zone in accordance with the "split rule."

Split rule: The split rule is a protocol. When sharing half a zone, the split peer splits its zone as a certain ordering of the dimensions [8].

For example, in a two-dimensional CAN, the zone is first split along its y-axis, then along its x-axis, and so on (see Figure 2.17).

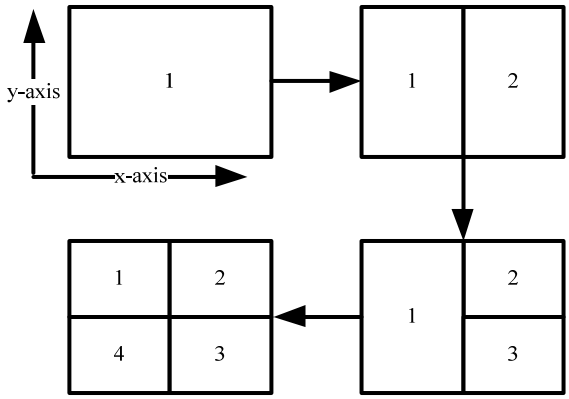


Figure 2.17 Split rule in a two-dimensional CAN

Coordinate routing table

During runtime, each peer maintains a coordinate routing table that consists of the IP address and zone boundaries of each of its immediate neighbors in the key space [6].

Neighbor: If two zones have overlapped coordinate spanning along *d*-1 dimensions and abutting along one dimension, they are neighbors in *d*-dimensional key space [6].

For example, Figure 2.17 illustrates that zone 1 is a neighbor of zone 4. As the definition of neighbor, zone 1 and 4 overlap along the x-axis and abut along the y-axis. In the other scenario, zone 4 is not a neighbor of zone 2 because two zones abut along both the x-axis and y-axis. Using the purely local neighbor state, the message can be transferred between two arbitrary points in the key space.

2.4.3 Peer Operation

As a new peer joins the system, it must be allocated its own portion of the key space. As a peer leaves, other peers have to take over its functionality immediately.

Joining

After retrieving a participating peer's IP address from the bootstrap peer, the new peer randomly chooses a point *p* in the key space and forwards a JOIN request to point *p*. The message is forwarded by peers according to the routing mechanism.

When the JOIN request arrives at the occupant peer (whose zone covers point p), the occupant peer splits its allocated zone in half, retains half and hands the other half to the new peer in accordance with the split rule [6]. Thus, the new peer obtains its own zone.

In order to join routing, the new peer copies the neighbor set from the previous occupant peer, and updates the neighbor set, i.e., adds the previous occupant peer and eliminates some peers that are no longer neighbors. Simultaneously, the previous occupant also updates its neighbor set. Finally, all neighbors are informed and asked to update their routing information.

During a new peer joining process, only the peers around the previous occupant is involved. In other words, the overhead when a new peer joins only depends on the number of neighbors and is independent of the size of the CAN. The average number of neighbors depends on the dimensionality of the key space. Hence, the overhead of a new peer joining will not scale up as the number of peers increase. The complexity is $O(\text{dimensionality})$.

Departure

When a peer leaves a CAN, it must ensure that its zone and the associated key-value database is taken over by the remaining peers. Therefore, the departing peer has to choose an occupant from its zone among its neighbors. If its neighbor peer *n*'s zone can merge with departing peer *m*'s zone and the merged zone is a valid hyper-rectangular zone, peer *m* should hand its zone over to peer *n*. Peer *n* eventually then extends its responsibility to take over *m*'s functionality and informs all neighbors to update their routing states. If any neighbor's zone cannot merge with the departing peer's zone, it has to hand its zone over to a neighbor that presently assumes the

smallest load. Simultaneously, the neighbor temporarily maintains two zones, i.e., the peer in CAN is allowed to handle more than one zone. For robustness and to avoid fragments, the peer with the multi-zone will try to hand over and merge its zone with its neighbor's zone. The process uses a zone-reassignment algorithm, and it is described in next section.

The CAN also needs a solution to peer or network failures. If a peer suddenly discovers that one or more peers are unreachable, the takeover mechanism is immediately triggered. Because it is possible that multiple adjacent peers are simultaneously involved in the failure region, first of all, it searches the region surrounding the failure region to ensure that more than half of the failed peer's neighbors are still reachable. If there are sufficient neighbors to initiate a takeover safely, each neighbor of the failed peer produces a TAKEOVER message conveying its own volume (e.g., load and quality of connectivity). Prior to sending the TAKEOVER message, they initialize a timer independently and wait for timer expiration. Once a peer receives a TAKEOVER message with a bigger volume than its own volume, it replies with its own TAKEOVER message. Otherwise, it cancels its timers and scrubs the TAKEOVER message. Eventually, an adjacent peer with the smallest volume is efficiently elected.

Second, the elected peer extends its responsibility to maintain the failed peer's zone, i.e., it temporarily handles a multi-zone. After a period of maintaining multiple zones, all peers eventually only handle one zone by means of the zone-reassignment algorithm.

In order to avoid stale key-value pairs as well as to recover lost key-value pairs, data holders periodically refresh their key-value pairs [6]. Thus, the peer or network failure causes the key-value pairs held by the crashed peer to be lost until the next refresh.

Zone-reassignment Algorithm

The departure procedure described in the foregoing section introduces the case of a single peer being assigned multiple zones. Consequently, CAN needs a zone-reassignment algorithm to retain the one-to-one peer to zone.

When a new peer joins CAN, the zone splits into two sub-zones. The zone is parent of the sub-zones into which it was split, and is termed the “partition tree” (see Figure 2.18). The partition tree records all splitting details from the beginning to the present. The leaf peers represent zones that presently exist in CAN (unshaded peers in Figure 2.18). The other peers represent zones that no longer exist, but had existed in the past (shaded peers in Figure 2.18) [62].

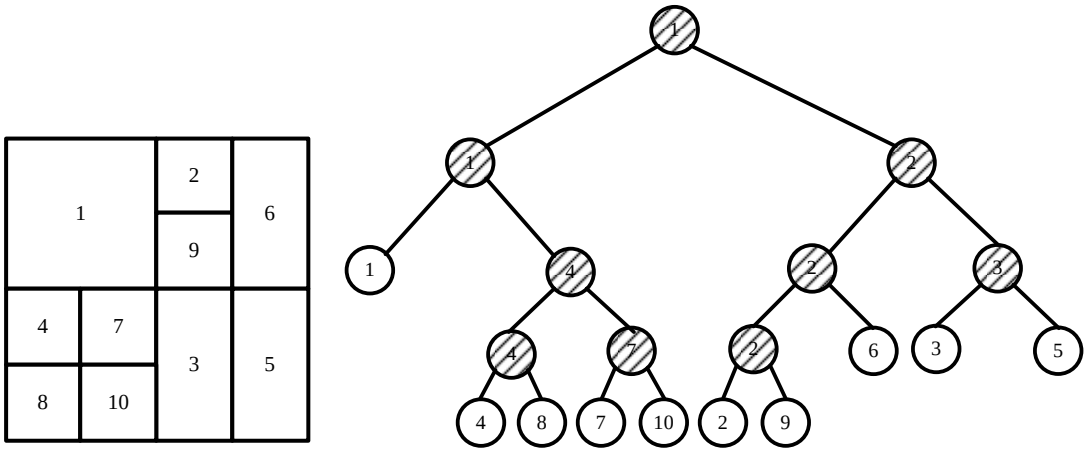


Figure 2.18 Partition tree

We utilize the same name for the peer in the partition tree corresponding to the zone. The partition tree is a binary tree. Only zones of sibling peers can merge with each other. If a peer *p* wants to hand its zone off, and *p*’s sibling peer *q* is not a leaf in the partition tree, *p* will depth-first search in the subtree of the partition tree rooted at *q* until it finds two sibling leaves (peers *m* and *n*): *m* extends its responsibility to merge with *n*’s zone, and then *n* occupies *p*.

However, to build a partition tree needs global view and space splitting history. It is almost impossible for a Peer-to-Peer system. We proposed our solutions in section 3 and section 4.

2.4.4 Routing

Each peer in CAN only stores the state of its immediate neighbors. Hence, each peer can only forward messages to its neighbor. CAN does not have any long links to send messages further over its neighbors. The routing algorithm is termed greedy routing. Greedy routing is a common algorithm in Peer-to-Peer systems. It is simple, but routing is not fast. Therefore Peer-to-Peer systems have some improved algorithms (finger table in Chord).

2.4.5 Design Improvements

The basic design of CAN provides $O(d)$ per-peer state in even zones CAN. In the worst case, a peer has $\frac{n}{2}$ neighbors (*n* peers in CANS). The routing complexity of *d*-dimensional CAN is $O(d \cdot n^{1/d})$ (*n* peers in CANS) [6]. The number of hops is not IP level but application level hops. Therefore, the distance between adjacent peers might be many miles and many IP hops. The average total number of hops is as follows:

$$L_{total} = n_{average\_number\_of\_CAN\_hops} \times L_{average\_latency\_of\_each\_CAN\_hop}$$

Design improvements are used to achieve smaller potential IP path latencies between the requester peer and the target peer.

Multi-dimensioned coordinate spaces

Each peer has $O(d)$ neighbors in *d*-dimensional CAN. If the number of neighbors per peer are increased by increasing the dimensions of the CAN key space, the average path length ($O(d \cdot n^{1/d})$) is shorter [6]. Simultaneously, each peer's routing table slightly increases. Figure 2.19 illustrates the effect of multi-dimensioned key

space. The simulations are based on Transit-Stub (TS) topologies and the GT-ITM topology generator [6, 63].

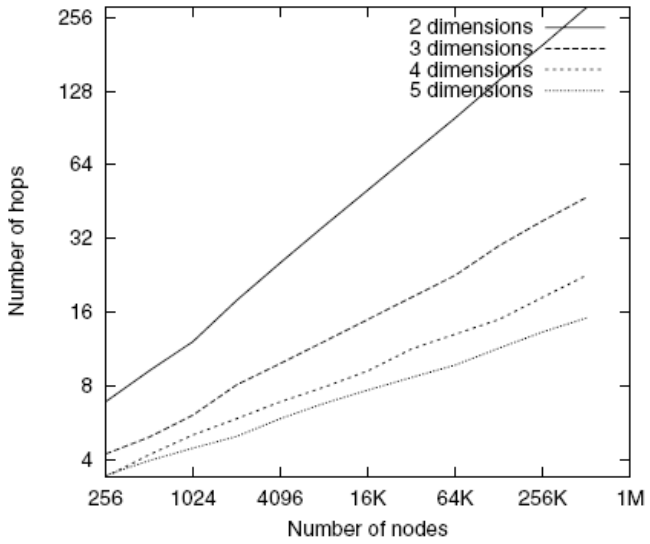


Figure 2.19 Effect of dimensions on path length [6]

Because each peer has more neighbors, the routing has more fault tolerance. Even though some adjacent peers have crashed, the current peer has more potential next hop peers to forward.

Realities: Multiple coordinate spaces

CAN maintains multiple key spaces independently. Each key space is termed a "reality." Each single peer in CAN is assigned multiple zones that are on distinct realities, i.e., each reality assigns one zone to each single peer. Simultaneously, each peer handles multiple independent neighbor sets in distinct realities.

For a key-value pair, we retrieve *r* coordinates ($(x,y)_r = \{(x,y)_1,(x,y)_2 \cdots,(x,y)_r\}$) in *r* realities by means of a hash function. It implies that there are *r* independent replications in CAN. The replications improve the peer's tolerance to failure. If peers crashed, other peers could recover system via crashed peers' replications. However, a

crashed peer cannot be recovered in a multiple crashes in which all replications are crashed too.

To forward a message, a peer checks its neighbors on each reality and forwards to the neighbor closest to the target. Thus, CAN reduces the path length using multiple realities [6] (see Figure 2.20).

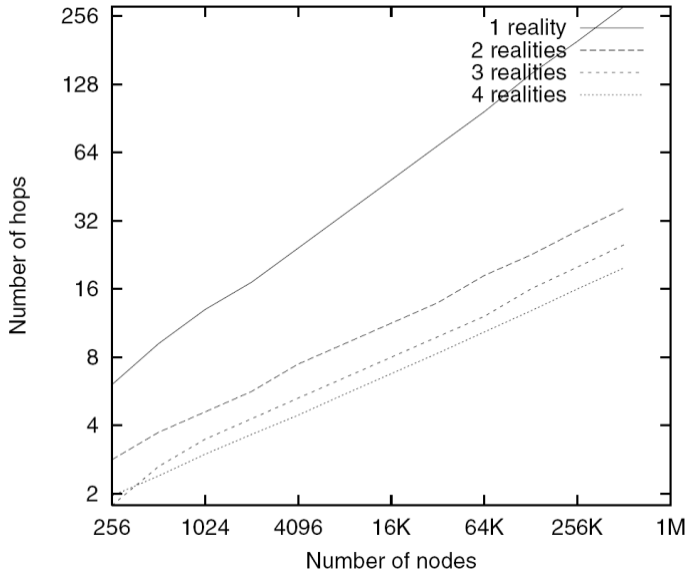


Figure 2.20 Effect of multiple realities on path length [6]

2.5 Routing improvement

Recently there are several works aimed at improving lookup efficiency in CAN

2.5.1 eCAN

eCAN is a mechanism proposed to enhance routing performance in “Building Low-maintenance Expressways for Peer-to-Peer Systems” [64]. The objective of eCAN is to establish a hierarchical scheme that maintains neighbor pointers at different levels of the logical space. eCAN operates similar to a real-world expressway. It improves CAN’s routing capacity by increasing the span of hops.

To establish expressways, the entire key space is divided into zones (see Figure 2.21). A set of zones assemble a bigger zone, which is termed “expressway zone.”. The span of the expressway zone k is a priori. A set of expressway zones assemble a higher-level expressway zone. Thus, the expressway zones have a hierarchical architecture. Figure 2.21 illustrates a two-dimensional space with k = 4. The region marked with dark shade is the zone, the region marked with green shade is a level-2 expressway zone, and the region marked with red shade is a level-1 expressway zone. Each zone in eCAN is a resident of the different level expressway zones, which enclose this zone.

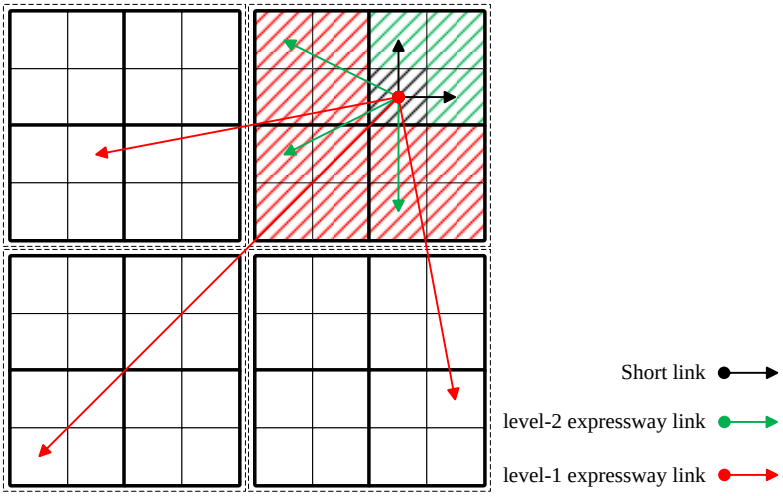


Figure 2.21 Expressways for CAN

Each expressway zone maintains links to other adjacent expressway zones at the same level. Consequently, the routing table of eCAN consists of not only the short links that link only to immediate neighbors, but also the long links that link to one peer in each of its adjacent expressway zones at different levels (different level links are marked with different colors).

Figure 2.22 is a snapshot of eCAN with k = 4. The expressway has established a binary tree, which is independent of the dimension of the key space. Consequently,

eCAN possibly achieves $O(\log n)$ routing performance by means of maintaining logarithmic routing entries at each peer in a CAN overlay

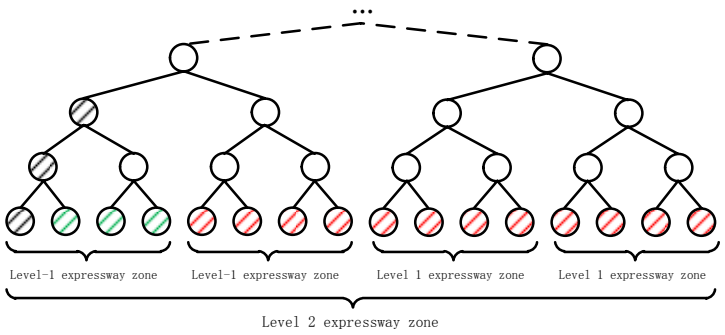


Figure 2.22 Snapshot of eCAN with k = 4

When peer leaves, its neighbor will take over its responsible zone. There are two scenarios. If the departure peer does not handle expressway zone, nothing extra needs to be done as conventional CAN. If this is not the case, the expressway is broken. There are two steps to repair a broken expressway in a demand-driven manner. First, when peer *s* forwards message to departure peer *d* via expressway, the routing request will time-out. Since eCAN is only an auxiliary system, the message can be forwarded using CAN greedy routing. Then, the recovery procedure is triggered. Peer *s* picks up a point in the zone of peer *d* and routes to it. Since peer *d*'s neighbor definitely took over its zone (CAN recovery operation), the routing will always succeed at peer *n* whose zone contains the point. Peer *n* must be a descendent of peer *d* and inherits peer *d*'s routing capability. Peer *n* replaces peer *d* to repair expressway.

2.5.2 LDPs

Each peer equips Long Distance Pointers (LDPs) [65] to add distant neighbors. Instead of greedy CAN routing, a peer considers both its short links and LDPs. A peer chooses the immediate/distant neighbor whose zone is closest to the target as the next hop. CAN with LDPs has a priori fixed routing state per peer, e.g., each peer keeps *k* LDPs. There are two different schemes for selecting LDPs:

Random Pointers

In this scheme, a peer sends *k* discover-messages to *k* random points in key space via greedy CAN routing. The peers, which cover those points, have the responsibility of replying with their IP addresses and zone information. The initiator establishes LDPs according to the replies. Figure 2.23 illustrates a peer *p* maintaining LDPs (*k* = 4).

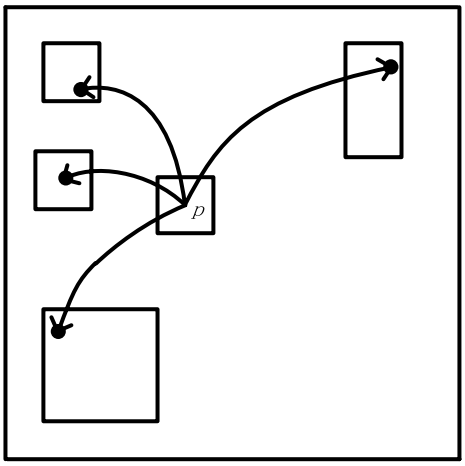


Figure 2.23 Random pointers

Distant neighbors in random pointers scheme might not be evenly distributed in key space. The example in Figure 2.23 has no pointer to the bottom right. In order to provide better coverage, subspace pointers are proposed.

Subspace Pointers

In contrast to random pointers, the subspace pointers scheme divides the key space into *k* equal-sized sub-zones, and each peer selects a random point from each sub-zone. The peers, whose zone covers the random point, are distant neighbors, and establish LDPs pointing to these distant neighbors.

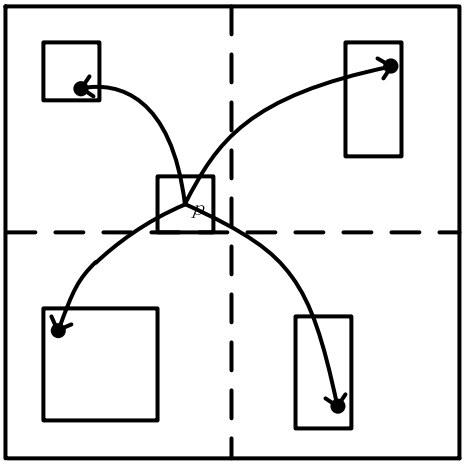


Figure 2.24 Subspace pointers

2.5.3 RCAN

"Multi-ring Infrastructure for Content-Addressable Networks" [66] was proposed as a novel topology to improve the routing efficiency of CAN overlays. In conventional CAN, a peer knows only about its immediate neighborhood. The greedy routing using only neighboring peers is not efficient and is more vulnerable to network failures. The key idea of RCAN is to equip each peer with long links towards some distant peer (called distant neighbors). Long links are established as follows:

"A node selects distant neighbors situated at distances inverse to powers of 2 on the coordinate space. The set of long links in each peer is partitioned into small sub-sets, each of which is established along one dimension. Long links are clockwise directed and wrapped around the key space."[67] The architecture of RCAN is a virtual multi-dimensional Cartesian space on a torus. (see Figure 2.25).

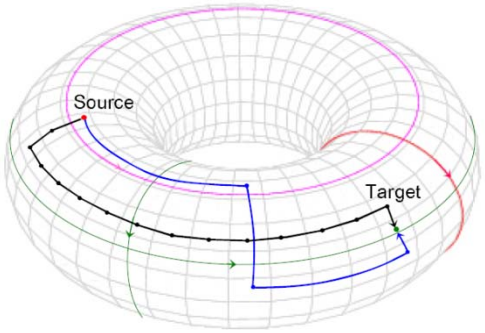


Figure 2.25 Torus model of a 2D multi-ring topology [66]

Each zone in RCAN associates with a set of positive integers $(l_0, l_i, \cdots, l_{d-1})$ that keeps track of the evolution of the key space—termed the region's level. l_i is the *i*-th sub-level of the zone. Its value implies the number of splits that the zone has undertaken along the *i*-th dimension. In order to eliminate inefficiency and vulnerability, each peer equips *d* sets of long links in d-dimensional space. Each set of long links points to distant neighbor peers located at distances inverse of the power of two from itself along one dimension. Consequently, a peer has $(l_i - 1)$ long links on the *i*-th dimension. The distance between a peer p and its *j*-th neighbor on the *i*-th dimension is

$$\left(p_i^j\right) = 2^j \cdot w_i$$

(w_i is the width of p's zone along the *i*-th dimension, and $j = 1, \cdots, l_i - 1$). Since the link for $j = 0$ is a short link pointing to an immediate neighbor (see Figure 2.26) [66].

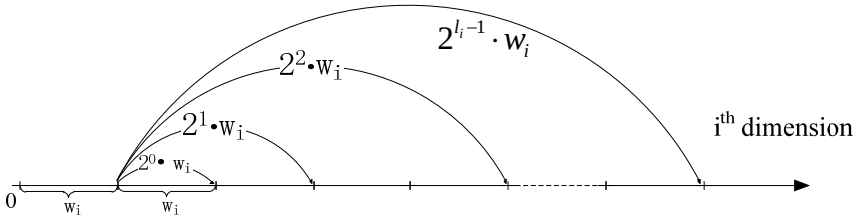


Figure 2.26 Long links in RCAN

Figure 2.27 illustrates the long links mode in two-dimensional RCAN.

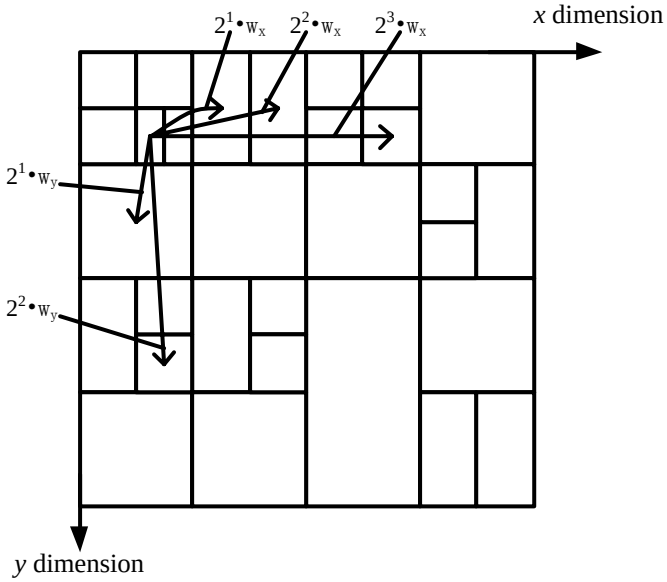


Figure 2.27 Long links model

RCAN adopts a “maintain-on-use” approach to update routing tables after peer activities. Instead of propagating the update to all affected peers, RCAN does nothing until a peer detects a link broken during routing. The peer can perform the routing task via other good links. Meanwhile, a process is triggered to fix the broken link. Hence, RCAN gets rid of high traffic overhead to inform all affected peers after peers churn [66].

RCAN provides a completely decentralized mechanism and self-scaling routing state. The number of long links per peer is $O(\log n)$ and maintenance overhead during peer churn is also $O(\log n)$.

Chapter 3

Using CAN Tree to Manage a CANS

In order to manage a CANS, we need the splitting history of CANS. CAN tree is a kind of data structure to record the splitting history.

3.1 Introduction

Content-Addressable Network for Distributed Simulations (CANS) is designed to handle the simulation of city traffic [68] or a Massively Multiuser Virtual Environment (MMVE). It is an improvement over normal CAN according to the purpose of the simulations. The two-dimensional simulation area is divided amongst the peers in CANS. Every peer handles exactly one zone. To a high degree, peers can run the simulation of their zone independently. When players or cars cross zone boundaries, a peer has to synchronize and hand them over to its neighbors. In order to improve efficiency, communication between peers must be reduced.

In simulations, we found that frequent communication between peers greatly reduces system efficiency. Since crossing of boundaries results in communication and concave zones causes a large number of unnecessary boundary crossings, concave zones are a problem. If a zone is concave, a car passing it may cross the boundary more than two times. In another scenario, when zones are slim, cars will cross the boundaries often. We call these two problems the “concave and slim” problem. We propose a new approach to solve them. All zones are convex and their length-width ratios will be limited to an acceptable range.

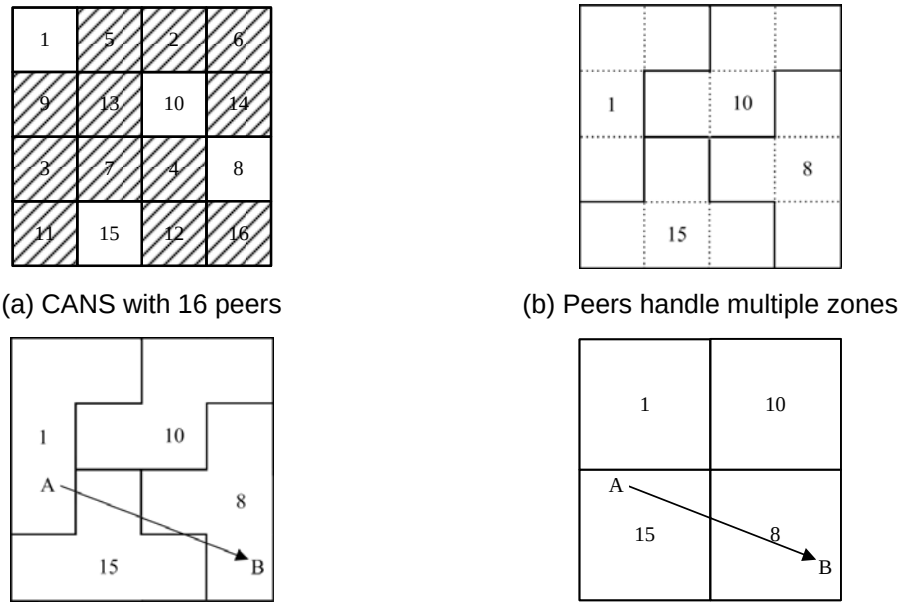
To eliminate concave and slim zones in CANS raises another problem. After a peer leaves, a left-recursion algorithm can be used to handle the zone released by the

departed peer. The number of recursive steps is unpredictable. In the worst case, all peers are involved. This will seriously affect the system's scalability and stability. We designed a "shortcut" algorithm that utilizes the "CAN tree" to solve this problem. Using our approach, recovery can be achieved in two steps, which is a significant speed up.

3.2 Requirements

When peers leave or crash, we must make sure that their zones are handled by other existing peers. After taking over the zones of other peers, a peer may have to handle multiple zones or handle a polygon zone. Since CAN is designed for file-sharing [6], these polygon zones are acceptable.

In contrast, CANS is designed for simulation. Although peers handling multiple zones are easy to realize, it generates extra load; and the extra load will never disappear until these zones can be merged. For example, CANS includes 16 peers, and every peer handles only one zone (see Figure 3.1(a)). When some peers leave (the departing peers are shaded in Figure 3.1(a)), the neighbors take over their zones. Eventually, every peer handles four zones (see Figure 3.1(b)). When a peer handles an increasing number of zones, the management of zones will become increasingly complex. Complex systems reduce efficiency and robustness, so CANS does not allow a peer to handle multiple zones.



(a) CANS with 16 peers (b) Peers handle multiple zones

(c) Peers handle concave polygon zones (d) Peers handle convex zones

Figure 3.1 Concave and convex zone

If zones are arbitrarily merged with zones that have been released by departing peers, a concave polygon may result. Figure 3.1(c) depicts a CANS after merging. Because the concave polygon zones increase the communication between peers, they incur extra costs. For example, we simulated city traffic with CANS. In the simulation, a car drove from location A to location B. In concave polygon zones (see Figure 3.1(c)), it crosses the boundaries four times. If the zones are convex (see Figure 3.1(d)), the car crosses the boundary only once. We try to keep CANS running with a simple structure and high efficiency; therefore, our peers must have neither multiple zones nor a polygon zone. We therefore enforce the rule that every peer handles exactly one rectangular zone. Zones cannot be arbitrarily merged. After merging, CANS must ensure that all zones are rectangles and their length-width ratios limited to an acceptable range.

3.3 Peer Churn

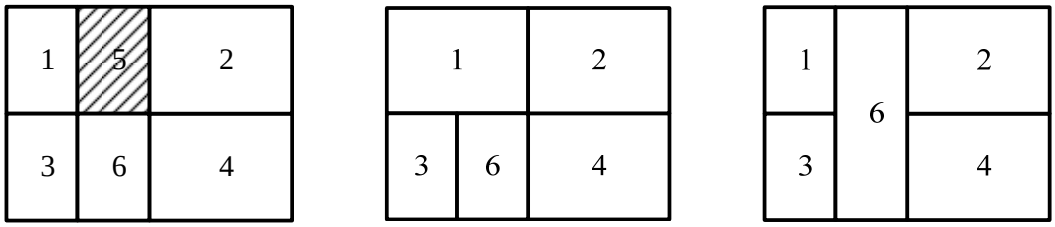
Peer joining and departure have to follow the protocols. The protocols make CANS stably run.

3.3.1 Peer joining

Because peer joining does not generate concave zone, joining a CANS network is done the same way as in CAN. At the beginning of CANS, only one peer handles the entire coordinate space. When a new peer joins CANS, it is allocated its own portion of the key space. This is done by an existing peer splitting its allocated zone in half, retaining half and handing the other half to the new peer [6]. In sharing a zone, the split peer splits its zone according to the "split rule", which is same with original CAN.

3.3.2 Peer Departure

When peers leave or crash, CAN must ensure that all zones are rectangles and their length-width ratios are limited to an acceptable range. We call this "acceptable." Sometimes, all zones are rectangles, but the layout of the zones is nevertheless unacceptable. For example, Figure 3.2(a), Figure 3.2(b), and Figure 3.2(c) show two layouts after peer 5 leaves. Here, the resulting state (i.e., zone split) cannot always be reached from the initial state according to the split rule.



(a) Peer 5 is leaving (b) Acceptable zones (c) Unacceptable zone

Figure 3.2 Acceptable and unacceptable zones

For example, zone 1 merges with zone 5 (see Figure 3.2(b)): this layout can be formed by splitting the entire key space according to the split rule. We call this layout "acceptable." If peer 6 merges with zone 5, the CAN becomes as shown in Figure 3.2(c). Although all zones are rectangular, this layout will never be formed by splitting the entire space according to the split rule. Therefore, we call this "unacceptable."

Definition 3.1: “Acceptable” is a state that is reached by splitting the entire space according to the split rule.

Definition 3.2: “Merge” is a zone action. When a zone of a peer merges with another, it extends its responsibility to take over the zone. After merging, CANS must be acceptable (see Figure 3.3).

Definition 3.3: (“Mergeable-zone,” “mergeable-sibling-zone,” and “mergeable-zone-pair”). When two zones can merge with each other, they are mergeable-zones: One is the other’s mergeable-sibling-zone. Both of them are a mergeable-zone-pair.

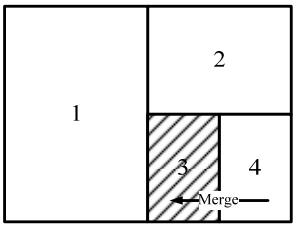


Figure 3.3 Merging

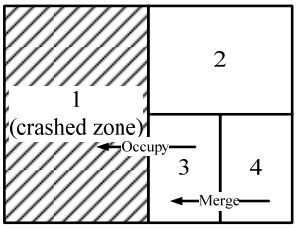


Figure 3.4 Occupying

Definition 3.4: “Occupy” is a zone action. A mergeable-zone peer releases its own zone and takes over the zone of another peer. The released zone will be merged by its mergeable-sibling-zone (see Figure 3.4). Only mergeable-zone peer can occupies others. Because occupier is mergeable-zone peer, the released zone of occupier definitely can merge with its mergeable-sibling- zone peer.

When a crashed zone cannot merge with its neighbor, we use “occupy” to handle the crashed zone[68]. A neighbor occupies a crashed zone and releases its own zone. Subsequently, other peers try to merge with the released zone, and so on. It is recursive. The recursion process will not be terminated until a mergeable-zone peer is found.

In CAN, all peers are independent and do not have a global view. The smallest neighbors of crashed peer[6] must occupy the crashed peer. Depending on the different neighbor to occupy, we get a different result and hence a different step count.

Therefore, the number of recursion steps is indeterminate. Figure 3.5 and Figure 3.6 show examples. The solution in Figure 3.5 uses five steps (four occupying and one merging), but the solution in Figure 3.6 uses only three steps (two occupying and one merging).

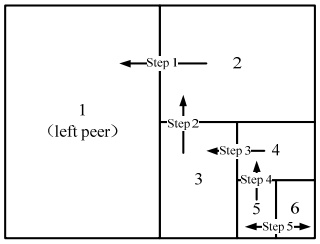


Figure 3.5 Long process of peer departure

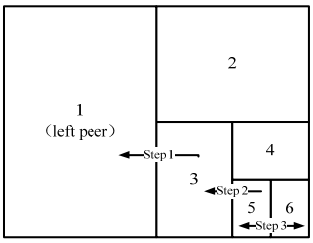


Figure 3.6 Short process of peer departure

The number of recursion steps is unpredictable. In the best-case scenario, a neighbor's zone can merge with the released zone. Here, only one step is required. However, in the worst-case scenario, we traverse almost all the peers. Here, we need as many steps as there are peers. This will seriously affect the system's scalability and stability.

In order to improve the worst case bound, we introduced an improvement scheme "shortcut." Our design does not restrict which peers can take part in the release-zone process. When a peer cannot find a mergeable-zone peer among its neighbors, it can directly ask a non-neighbor mergeable-zone peer to handle it. In this scenarios, CANS needs additional links. We call it "CAN tree" and introduce it later.

Because a mergeable-zone can merge with its mergeable-silbing-zone, the process uses only two steps. In the best-case scenario (merge with a neighbor), we nevertheless need one step. However, in the worst-case scenario (merge with a non-neighbor), we need only two steps. Figure 3.7 shows the path of "shortcut-release-zone." Peer detects the states of its neighbors by means of periodical heartbeat messages [66]. When a neighbor of peer 1 detected peer 1 crashed, it informs all neighbors of the crashed peer[6]. It will be in charge of recovery. It found a mergeable-zone-pair peer 5 and 6, and asks peer 5 to occupy the zone of peer 1. Peer 6's zone then merges with the zone released by peer 5.

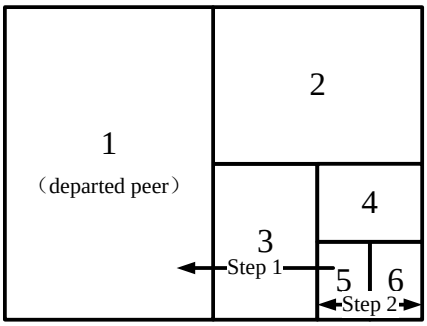


Figure 3.7 Shortcut peer departure

In the “shortcut release zone,” the key is to find a mergeable-zone in CAN. We designed the “CAN tree” to search mergeable-zone. CAN tree is a data structure that records the splitting history. Figure 3.8 shows a CAN and its CAN tree. The shaded zones are mergeable-zones. Every peer represents a zone. Using the CAN tree, we can easily find these mergeable-zones.

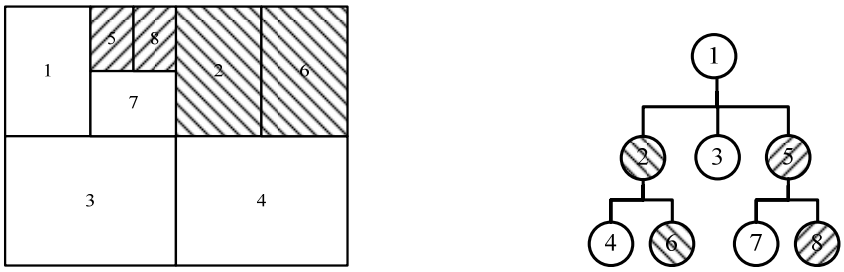


Figure 3.8 CAN and its CAN tree

3.4 CAN Tree

In order to find mergeable-zones, we need the splitting history of the key space. We use tree structure to record splitting history, it’s termed “CAN tree.”

3.4.1 Building a CAN Tree

When peers join CAN, an existing peer split its zone in accordance with the split rule. At the same time, the “parent-child” (long link) relation between peers is established[68]. If new peer *p* got half a zone from peer *q*, *q* becomes the parent of *p*. All “parent-child” relations comprise a tree[68].

In the beginning, there is only one peer in CAN to handle the entire coordinate space. The peer is the root of the tree. As new peers join CAN, the CAN tree grows. Figure 3.9 shows how a CAN tree is built. The shade zones are mergeable-zones.

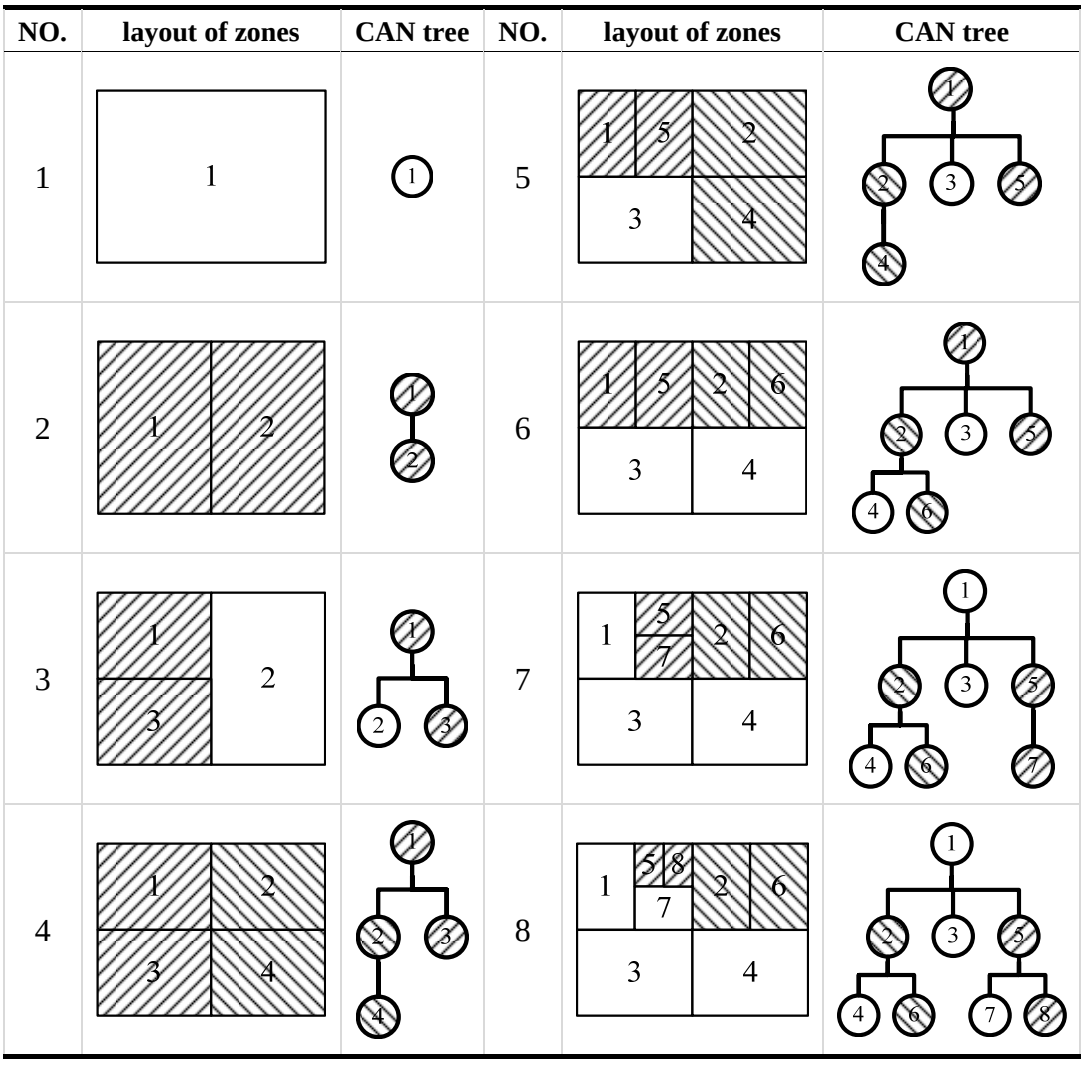


Figure 3.9 Building a CAN tree

The CAN tree looks similar to partition tree. However, they are different. In partition tree, the leaf peers exist in CAN. The other peers no longer exist, but have existed in the past[56]. Thus, global view and space splitting history are necessary to build a partition tree. In contrast, every peer in CAN tree exists in CANS. Peers only need to know their parents and children. The global view is not necessary.

3.4.2 Storing the CAN Tree

There are multiple options for storing the CAN tree. The most straightforward solution is to store the CAN tree on a dedicated server or to select a peer (or set of

peers) that is responsible for storing the tree. This CAN tree server records all relationship between peers. It is simple. But it has bottle neck problem. It also becomes the weakness of CANS. If this server crashed, CANS cannot work anymore. The second option is to build a distributed CAN tree: Every peer records a set of pointers that build connections between parent peer and child peer (see Figure 3.10). When peers join or leave (actively) CANS, we only modify the pointers on the affected peers. When a peer crashed (passively), one of its neighbors will lead recovery (introduce later in section 3.4.3). It also inform the involved peers to modify the pointers. The peer churn then affects only a small number of peers. Because updating pointers is independent of the number of peers in the system, the cost of the operation is constant. Therefore, the cost of the CAN tree is very cheap and CAN can scale well. Traversing the tree becomes more costly when the tree is distributed instead of centralized. However, with a decentralized storage there is no risk of losing the tree if the dedicated tree server peers crash all together.

(a) CANS (b) CAN tree

Figure 3.10 Distributed CAN tree

3.4.3 Finding Mergeable-Zones

If a zone is split into two sub-zones and the sub-zones are not split any more, the two sub-zones are a mergeable-zone-pair. According to this rule, we could judge whether two zones are a mergeable-zone-pair.

Because the order of sibling nodes (from left to right) is the order in which a zone is gained from the parent, the youngest node in the sibling nodes is the rightmost node. For example, Figure 3.8 shows node 5 having two child nodes. Node 8 is the

rightmost leaf node. Therefore, the last split of zone 5 splits half of its zone to create zone 8. Zones 5 and 8 are then not split anymore.

Consequently, zones 5 and 8 are mergeable-zone-pair. We call this the "mergeable-zone-pair rule." We deduced that two zones that map to the rightmost leaf node and its parent node in a CAN tree, are a mergeable-zone-pair.

Peers use heartbeat message to test whether its neighbors crash. When a peer finds its neighbor crashed, it will inform all the neighbors of crashed peer. And it takes responsibility to lookup mergeable-zone-pair. We call it "leading peer". It checks whether the crashed peer is mergeable-zone. If it is, the leading peer send message to the mergeable-sibling-zone to merge with the crashed peer. Otherwise, the leading peer is in charge of search mergeable-zone-pair in CAN tree.

If the leading peer has children in CAN tree, it sends the message to its rightmost child (the youngest child) in CAN tree. For example in Figure 3.11, peer 1 is leading peer. It will send message to peer 5. Every peer received the message will send this message to its rightmost child until find a rightmost leaf peer. The leaf peer and its parent are mergeable-zone-pair. The rightmost leaf peer will take over the zone of crashed peer and release its own zone (it will be merged by its mergeable-sibling-zone.). The other neighbors of the crashed peer will not take part in the search.

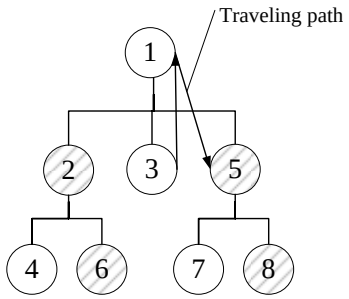


Figure 3.11 Traveling in CAN tree

If the leading peer has no child, it must send message to its parent peer in CAN tree. For example in Figure 3.11, the leading peer 3 sent a message to peer 1. Peer 1 sent a message to its rightmost child.

In multiple crashes, we maybe find a crashed peer in traveling path. For example in Figure 3.11, peer 3 is leading peer. It sent a message to peer 1. But peer 1 crashed too. In this case, peer 3 will find out the replication of peer 1(Original CAN reality solution in section 2.4.5). According to the replication of peer 1, peer 3 directly sent a message to the rightmost child of peer 1. In other word, traveling could jump over crashed peer via peer replication. Therefore, the crashed peers could be merged or occupied one by one.

By repeatedly traversing the rightmost child, we find the rightmost leaf node in any case. Using the CAN tree, we only need to traverse a small portion of peers to find a mergeable-zone. Therefore, our algorithm will always succeed and it significantly speeds up the process of reallocating zones.

3.4.4 Complexity of Searching Mergeable-Zones

We know that we can find a mergeable-zone by traversing the rightmost. Therefore, the complexity of searching a mergeable-zone depends on the peers count and the structure of the CAN tree.

The worst-case scenario

In the worst-case scenario, the CAN tree degenerates into a list structure, i.e., there is only one associated child node for each parent node. This means that in a performance measurement the CAN tree will essentially behave like a linked list data structure. Therefore, the complexity in the worst-case scenario is $O(n)$; and we plot the average length of the traversing path for an increasing number of nodes (the curve $f_{worst}(n)$ in Figure 3.12).

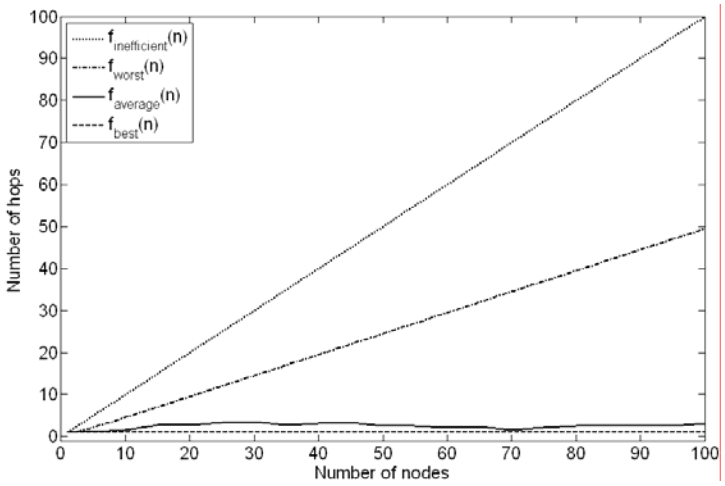


Figure 3.12 Effect of CAN tree searching

The best-case scenario

All zones are mergeable-zones (see Figure 3.13). Therefore, the length of the traversing path is $f_{best}(n)=1$ (the curve $f_{best}(n)$ in Figure 3.12). The complexity in the best-case scenario is $O(1)$.

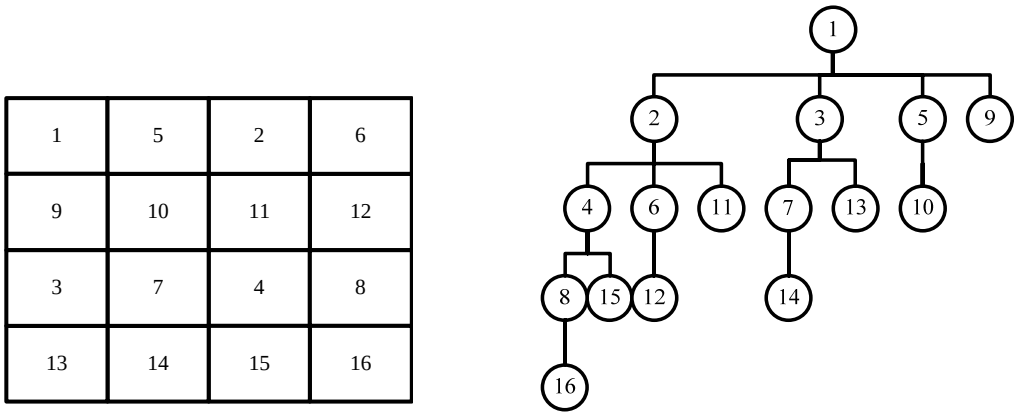


Figure 3.13 The best case

Average case scenario

In the previous cases, the length of the traversing path can be directly inferred from the algorithm; in the average case, we resorted to simulation. In simulation, we randomly generated 100 CAN trees. The number of peers ranged from 1 to 100. The simulation randomly makes a zone crash and calculated the length of the traversing path in every CAN tree. After running the simulation 1000 times, we got the average

length of the traversing path in CAN trees with different number of peers. We plot it for an increasing number of peers (the curve $f_{average}(n)$ in Figure 3.12).

The inefficient recursive algorithm searches mergeable-zones via the layout of CANS (i.e., it does not use the CAN tree), so it must visit all peers—resulting in a complexity of $O(n)$. Even if the CAN tree is in the worst-case scenario, it does not need to visit all peers. It only needs a depth-wise search to find the mergeable-zone, and does not care about nodes above the current node in the CAN tree. Although the inefficient algorithm and CAN tree searching in the worst-case scenario have the same complexity ($O(n)$). The inefficient algorithm is by a constant factor slower than the CAN tree in the worst-case scenario. (The curve $f_{inefficient}(n)$ in Figure 3.12.)

From Figure 3.12, it can be seen that our algorithm greatly reduced the path length. Furthermore, $f_{average}(n)$ is almost independent of node number. Therefore, we can avoid unnecessary hops via CAN tree, and CANS becomes more efficient.

3.4.5 CAN Tree Modification on Peer Departure

When a peer crashes or leaves, other peers will merge with or take over its zone, as discussed in section 3.3.2. We must modify the CAN tree according to the CANS modification. The CAN tree records the splitting history of the updated CANS; for example, as follows:

- Merging: At the beginning, the key space and CAN tree is as shown in Figure 3.14. Peer 6 is leaving. Because peer 6 is a mergeable-zone, it can be merged with its neighbor's zone. After peer 6 has left, peer 2's zone is merged with peer 6's zone (see Figure 3.15). In the CAN tree, node 6 is deleted.

Figure 3.14 CANS and CAN tree before departure

Figure 3.15 CANS and CAN tree after merging

- Taking over: When peer 4 is leaving (see Figure 3.14), its zone cannot be merged with its neighbor's zone. Peer 8 merged its zone with peer 5's zone. Peer 5 took over peer 4's zone (Figure 3.16). In the CAN tree, node 5 is transferred to the position of node 4 and node 8 is transferred to the position of node 5.

Figure 3.16 CANS and CAN tree after occupation

3.5 Conclusion

In this chapter, we presented CANS, a Peer-to-Peer network for conducting traffic simulations or MMVE games. CANS is based on CAN, but it features several optimizations that make it more useful in our simulation scenarios. CAN has some issues that are related to the shape of its zones: specifically, they can become arbitrary polygons. For file sharing, this is not a drawback, but for simulations, the communication overhead is lowest for quadratic zones.

Therefore, CANS uses a new algorithm to reallocate zones, such that they are neither concave nor slim. We have shown that a simple and straightforward solution can achieve the desired zone splitting, but too many peers are involved in swaps.

As a result, we introduced the CAN tree. This tree structure allows us to find mergeable-zones very efficiently. This greatly reduces the number of zone swaps between peers when compared to the simple approach. We showed that we can give a constant boundary for the number of swapping steps.

Chapter 4

Using Zone Code to Lookup Mergeable-zones

A CAN tree records the splitting history and allows to search mergeable-zones efficiently. However, peers need to communicate in order to modify the CAN tree after peers churn. We introduced the zone code to do the same work with the CAN tree, with no communication needed among the peers. Consequently, the zone code scheme reduces communication overhead.

4.1 Partition tree

When a new peer joins CANS, a zone is split into two sub-zones. If we record that the zone is the parent of the two sub-zones, the splitting history is a binary tree (see Figure 4.1). People call it “partition tree” (section 2.4.3). The partition tree records all the splitting details from the beginning to the present. The leaf nodes represent zones that exist in CAN. The other nodes represent zones that no longer exist, but have existed in the past[56].

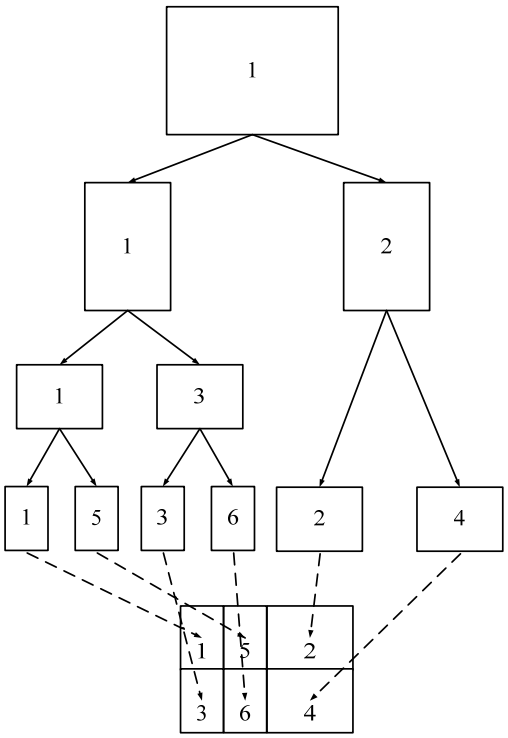


Figure 4.1 Partition tree

If a zone is split into two sub-zones and the sub-zones are not split any more, the two sub-zones are a mergeable-zone-pair. Combining all our insights, we deduced the following corollary:

Corollary 4.1: Two sibling leaf nodes in the partition tree, which share a common parent node, are mergeable-zone-pair peers.

Thus, the partition tree is a solution to search the mergeable-zones. Because the internal peers no longer exist (they existed at some previous time),, we can not use this tree structure. Instead of the partition tree, we only store the partial useful splitting history by zone code.

4.2 Building Zone Code

We do not store the entire partition tree in any peer, but rather convert the partition tree into bit sequences called the zone code (see Figure 4.2). Each zone stores a zone code (a part of partition tree). A zone code is a unique code, and it is fully

decentralized. A traversal of the partition tree is performed to obtain the zone code (see Figure 4.2). It is analogous with Huffman code [69]. Going left is a “0,” going right is a “1.” A zone code is completed when a leaf node is reached. We deduced the following corollary:

Corollary 4.2 The zone code is a binary prefix code.

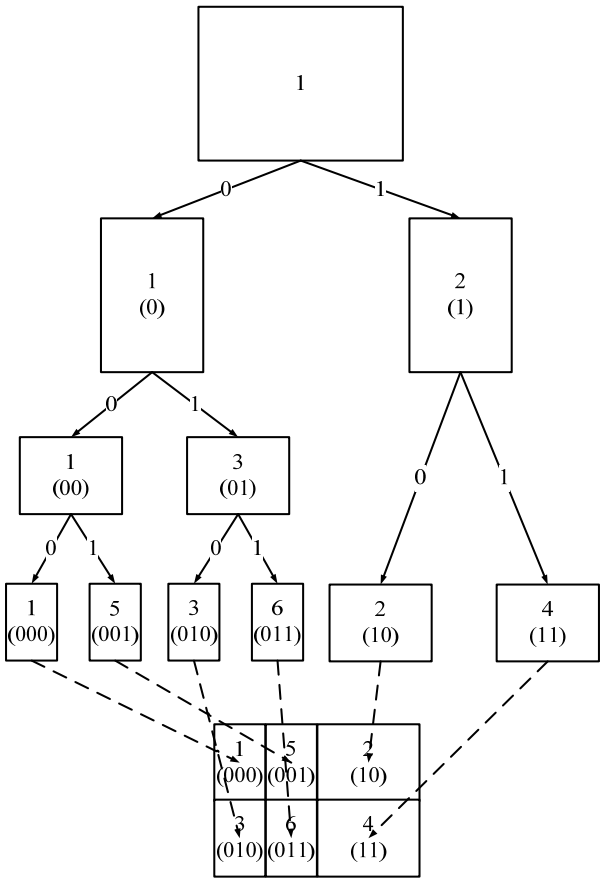


Figure 4.2 Zone code

In practice, we do not need the partition tree to generate a zone code. Every bit of the zone code signifies a split. The more CAN splits, the longer the zone code becomes. The zone code grows with splitting and shrinks with merging. When a peer joins or leaves, we only need to append or delete, respectively.

4.2.1 Zone code growth as peers join

When a new peer joins in CAN, an existing zone splits into two sub-zones. It retains one and hands the other one to the new peer. We append “0” to the zone code of one sub-zone, and append “1” to the other. After each splitting, the zone codes of the sub-zones will append 1-bit. Hence, the zone code grows simultaneously with splitting.

Figure 4.3 shows how the zone code grows during the key space splitting. Initially, a peer handles the entire key space and its zone code is null.

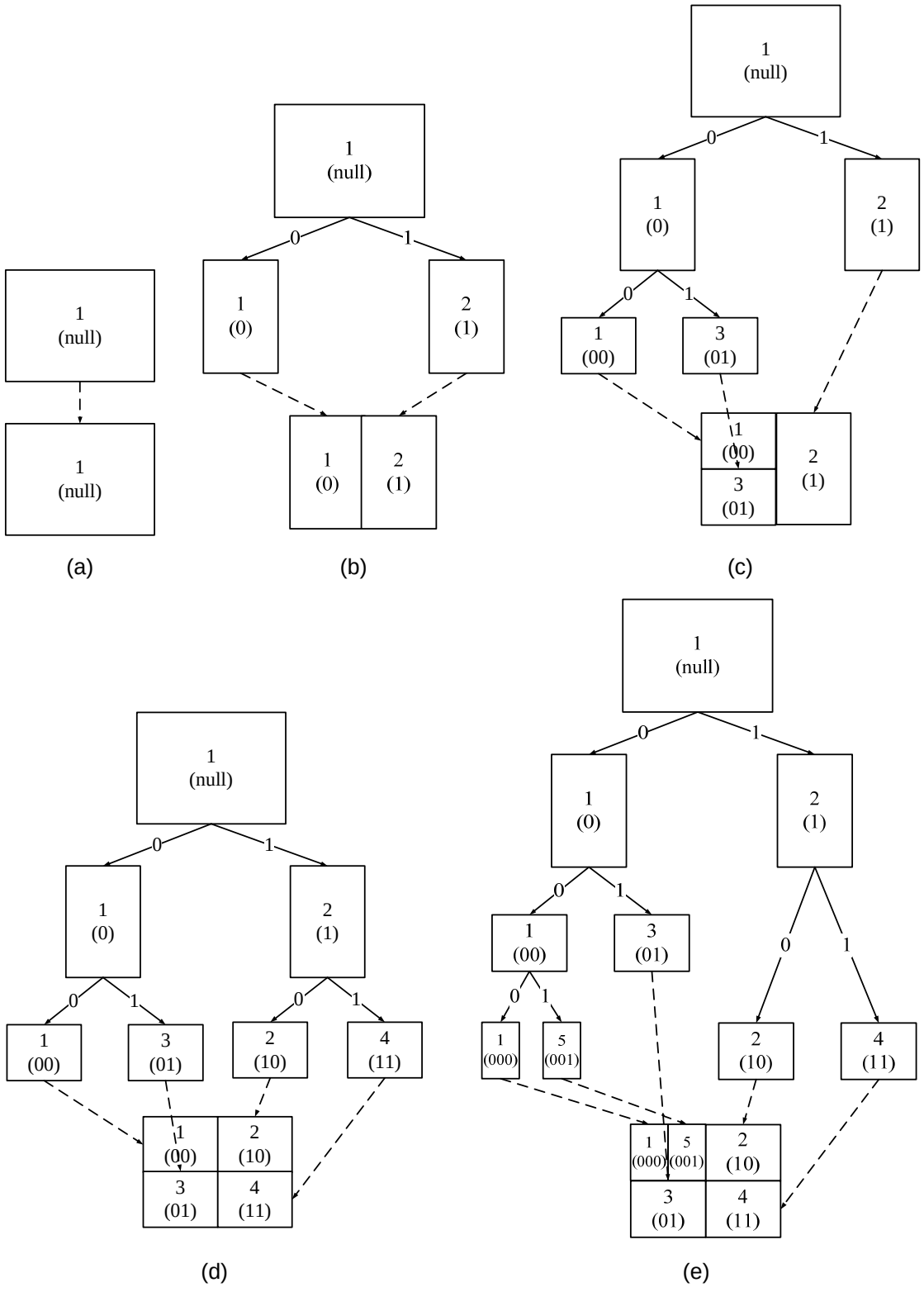
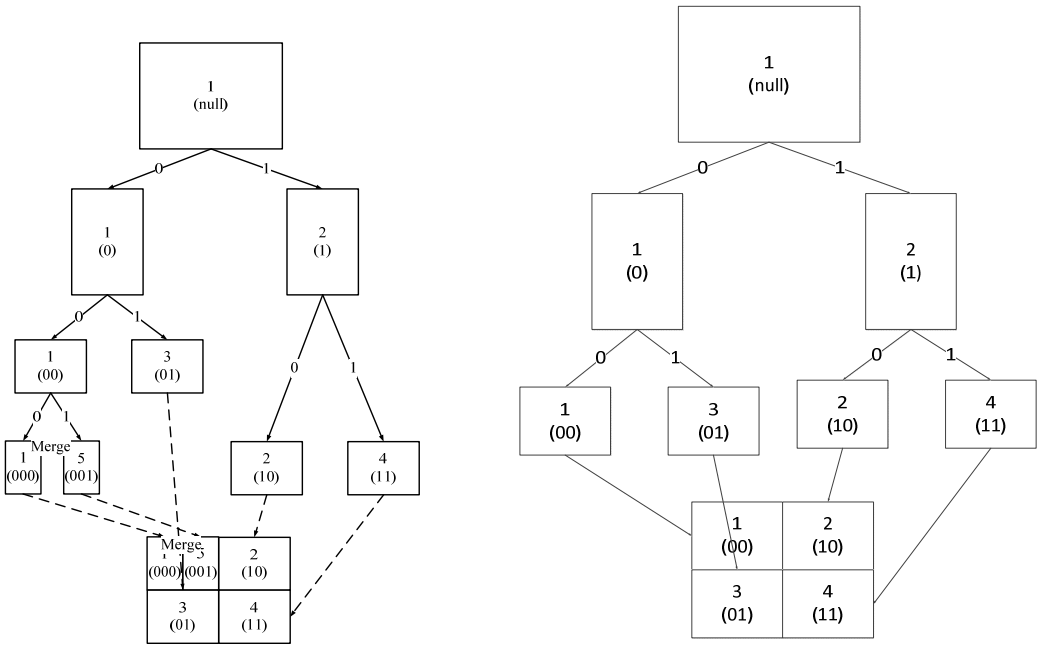


Figure 4.3 New peer joining

4.2.2 Zone code decrease as peers' zones merge

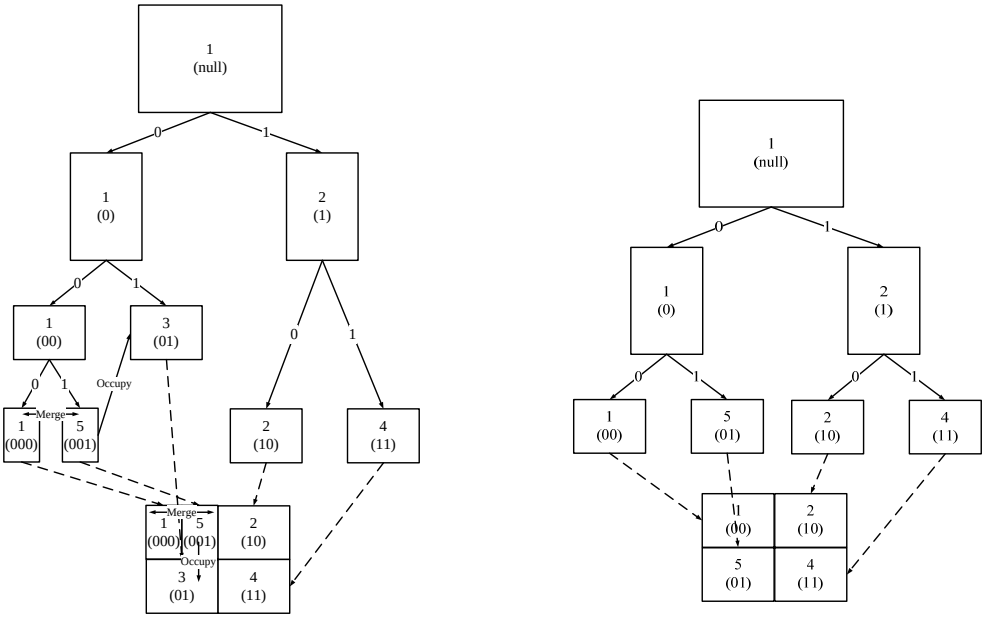
When a peer leaves or crashes, there are two resulting scenarios. If it has a mergeable-zone, its zone can merge with its mergeable-sibling-zone (zones 1 and 5 in Figure 4.4(a)). After merging, we delete the last bit of the zone code. For example (see Figure 4.4(b)), peer 1's zone merged with peer 5's zone, and the zone code changed from "000" to "00."



(a) Before merging (b) After merging

Figure 4.4 Peer departure and merging

In the other scenario, if the departed peer had a non-mergeable-zone, a non-neighbor mergeable-zone will occupy it (see Figure 4.5). After occupation, the zone copies the zone code of the occupied zone. For example, in Figure 4.5(b), peer 3 has crashed. Peer 5 then occupied peer 3, and released its zone. In addition, peer 1's zone merged with the zone released by peer 5, and deleted the last bit of its zone code (changed from "000" to "00").



(a) Before occupation (b) After occupation

Figure 4.5 Peer departure and occupation

4.3 How to find Mergeable-zones

Theoretically speaking, it is easy to search for a mergeable-zone-pair via the partition tree. The sibling leaf nodes that share a common parent node are a mergeable-zone-pair. Thus, we deduced the following corollary:

Corollary 4.3 Let $x=(x_1, x_2 \cdots x_{n-1}, x_n)$ and $y=(y_1, y_2 \cdots y_{n-1}, y_n)$ denote the two zone codes of zones X and Y. Then, $\forall i \in \{1 \cdots n-1\}: x_i \oplus y_i = 0 \wedge x_n \oplus y_n = 1 \Leftrightarrow$ X and Y are a mergeable-zone-pair.

In other words, if two zone codes differ only in the last bit, they are a mergeable-zone-pair. Corollary 4.3 is crucial to our solution. We do not need any more data structures to record splitting history, such as partition tree or CAN tree. Using logical operation exclusive disjunction, we can know whether two zones are a mergeable-zone-pair. For example, peer 1's zone code is "000" and peer 5's is "001" (see Figure 4.2). Because of $000 \oplus 001 = 001$, they are a mergeable-zone-pair. Peer 6 has zone code "011" and $011 \oplus 001 = 010$. Thus, peers 6 and 5 are not a mergeable-zone-pair.

4.4 Search Algorithm

By means of the zone code, we can determine whether two zones are a mergeable-zone-pair. However, when we need a non-neighbor mergeable-zone, we have to search in the distributed system. Therefore, we need an efficient search algorithm.

4.4.1 Area search

We start to search from a randomly chosen zone, which could be any zone. It is termed “starting zone”. First, we search the mergeable-sibling-zone of starting zone. It is termed “search area” (see Figure 4.6). If there is only one peer in the mergeable-sibling-zone of starting zone, it means that it did not split again. The starting zone and search area are mergeable-zone-pair (the starting zone is shaded in Figure 4.7(a))..Otherwise, there are two scenarios:

- Scenario 1: There is more than one zone in the search area and there is a mergeable-zone-pair among the neighbors of the starting zone. Because every zone directly communicates with its neighbors in CANS, we can get the zone codes of the neighbors of the starting zone. Hence, we can definitely find the mergeable-zone-pair by zone code (the starting zone is shaded in Figure 4.7(b)).
- Scenario 2: There is more than one zone in the search area and no mergeable-zone-pair among the neighbors of the starting zone (the starting zone is shaded in Figure 4.7(c)). We need to randomly choose a new starting zone in the search area and repeat this process, until we find a mergeable-zone-pair.

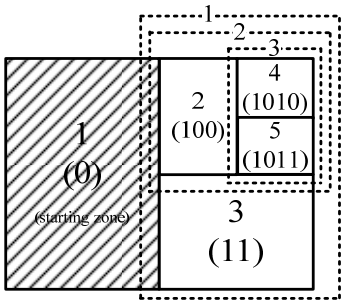
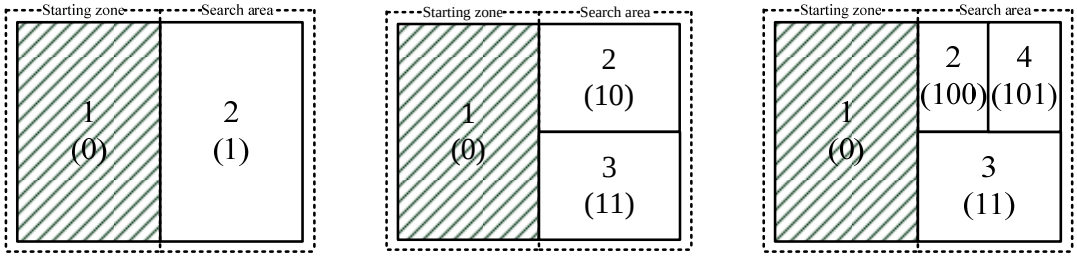


Figure 4.6 Shrinking search area



(a) One zone in the search area

(b) Mergeable-zone-pair in the search area

(c) No mergeable-zone-pair in neighbors

Figure 4.7 Three search scenarios

Because every zone splitting will generate a new mergeable-zone-pair, even if the search area is divided into sub-zones, there is definitely at least one mergeable-zone-pair in the search area. Searching is a recursive process to shrink the searching area. Therefore, our algorithm is always valid. Every step the search area shrinks half space at lest. It's a kind of binary search.

4.4.2 Complexity of Searching

We know that we can find a mergeable-zone by shrinking the search area. Hence, the searching complexity depends upon the shrinking rate.

The worst-case scenario

There is only one mergeable-zone-pair in CANS. Therefore, the complexity in the worst-case scenario is $O(n)$. The curve $f_{worst}(n)$ in Figure 4.8 shows the hop count with increasing peers.

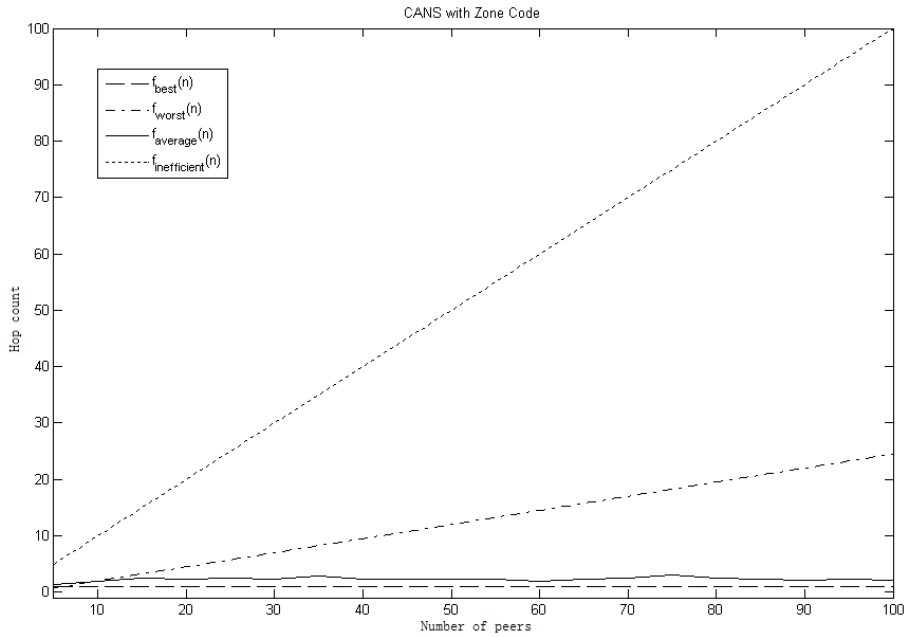


Figure 4.8 Effect of shrinking search

The best-case scenario

All zones are mergeable-zones. Therefore, the hop count is $f_{best}(n)=1$(the curve $f_{best}(n)$ in Figure 4.8). The complexity in this best case is $O(1)$.

Average case scenario

In the previous cases, the length of the traversing path can be directly inferred from the algorithm; in the average case scenario, we resorted to simulation. In the simulation, we randomly generated 100 CANS. The number of zones ranged from 1 to 100. The simulation then randomly chooses a peer to crash and calculated the length of the traversing path in every CANS. After running the simulation 1000 times, we got the average length of the traversing path in the CAN trees with different numbers of zones. We plotted it for an increasing numbers of peers for coordinate spaces: Figure 4.8 shows the curve $f_{average}(n)$.

The inefficient algorithm searches mergeable-zones via the layout of CANS (i.e., it needs global view.), so it needs a system snapshot. Hence, it has a complexity of $O(n)$

(n is the number of peers in system). When our algorithm is in the worst case scenario, it still does not need to visit all zones. Although the inefficient algorithm and our algorithm in the worst-case scenario have the same complexity, $O(n)$. The inefficient algorithm is by a constant factor slower than our algorithm in the worst-case scenario. Therefore, the length of traversing with the inefficient algorithm is $f_{inefficient}(n) = n$ (Figure 4.8, the curve $f_{inefficient}(n)$).

From Figure 4.8, it can be seen that our algorithm greatly reduces the search complexity path length. Furthermore, $f_{average}(n)$ is independent of the peer count. Therefore, we can avoid unnecessary hops and CANS becomes more efficient. Figure 4.9 illustrates the search complexity path length distribution of CAN tree with 100 peers. More than 45 percent only need 1 step to find mergeable-zone-pair.

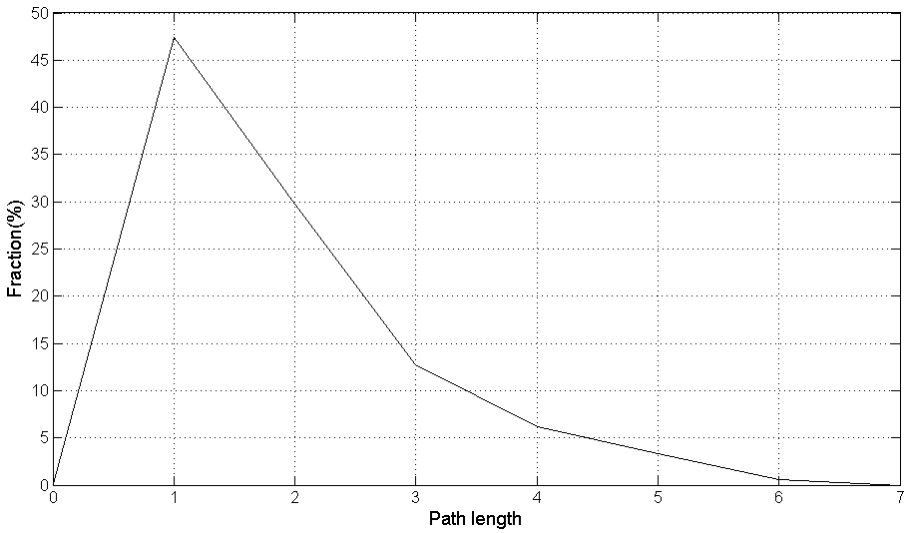
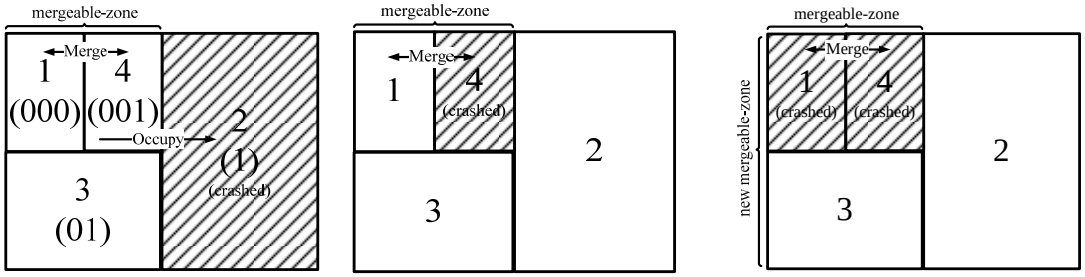


Figure 4.9 Search complexity path length in shrinking search

4.5 Multiple Crashes

After a new peer joins CANS, it and its zone supplier are become a "mergeable-zone-pair." Thus, CANS has at least one "mergeable-zone-pair." When peers crash, we encounter three kinds of scenarios:

- Scenario 1: CANS has at least one valid mergeable-zone-pair whose sibling zone is not crashed (see Figure 4.10(a)).
- Scenario 2: CANS has only valid mergeable-zone, whose sibling-zone has crashed (see Figure 4.10(b)). There is at least one such mergeable-zone.
- Scenario 3: CANS has no valid mergeable-zone (see Figure 4.10(c)). All mergeable-zones have crashed.



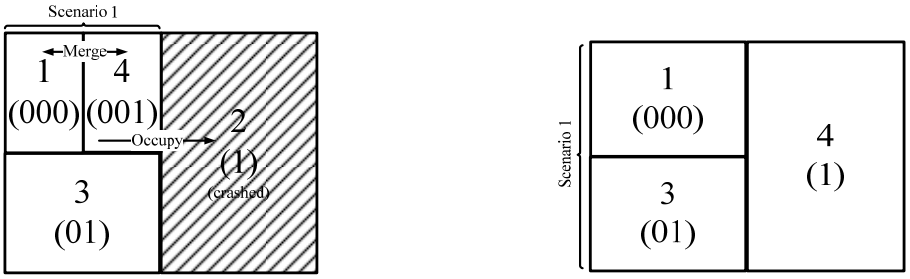
(a) Scenario 1 (b) Scenario 2 (c) Scenario 3

Figure 4.10 Three scenarios in multiple crashes

In multiple crashes, our strategy is that a valid zone merges with the crashed zone and takes over its responsibility. However, after merging, CANS may change from one scenario to another. In this section, we propose different algorithms for different scenarios.

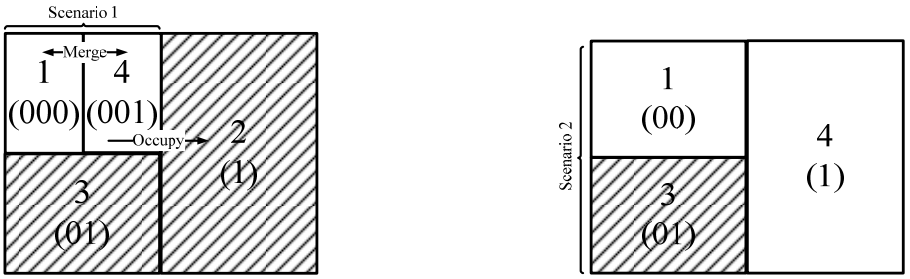
Algorithm for scenario 1

In this scenario, there are at least two valid mergeable-zones. One occupies the crashed zone. Its sibling-zone merges with it. In this step, one crashed zone is recovered. Then scenario 1 may change into scenario 1, 2, or 3 (see Figure 4.11, Figure 4.12, and Figure 4.13).



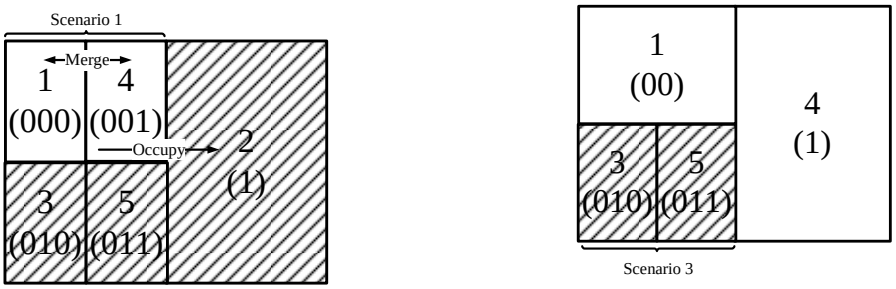
(a) Before occupation (b) After occupation

Figure 4.11 Scenario 1 to scenario 1



(a) Before occupation (b) After occupation

Figure 4.12 Scenario 1 to scenario 2

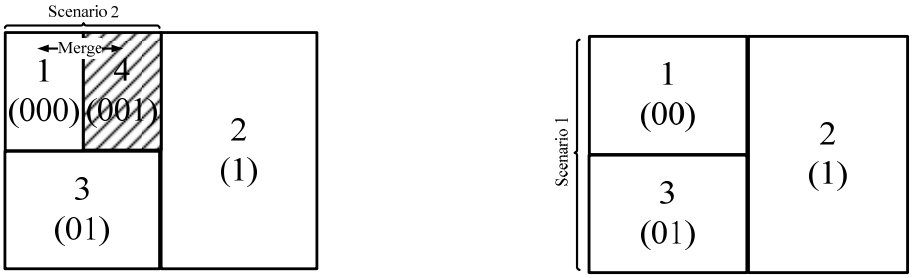


(a) Before occupation (b) After occupation

Figure 4.13 Scenario 1 to scenario 3

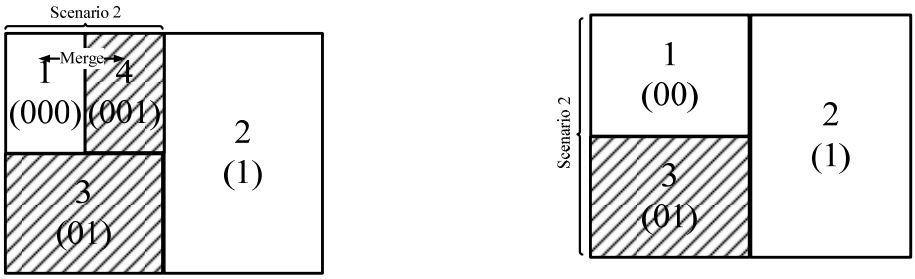
Algorithm for scenario 2

In this scenario, there is only one valid mergeable-zone, and its sibling-zone has crashed. The valid zone merges with its crashed sibling-zone. In this step, one crashed zone is recovered. Scenario 2 may then change into scenario 1, 2, or 3 (see Figure 4.14, Figure 4.15, and Figure 4.16).



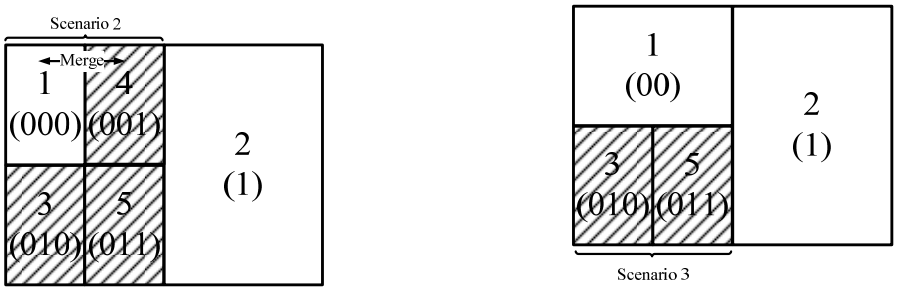
(a) Before merging (b) After merging

Figure 4.14 Scenario 2 to scenario 1



(a) Before merging (b) After merging

Figure 4.15 Scenario 2 to scenario 2

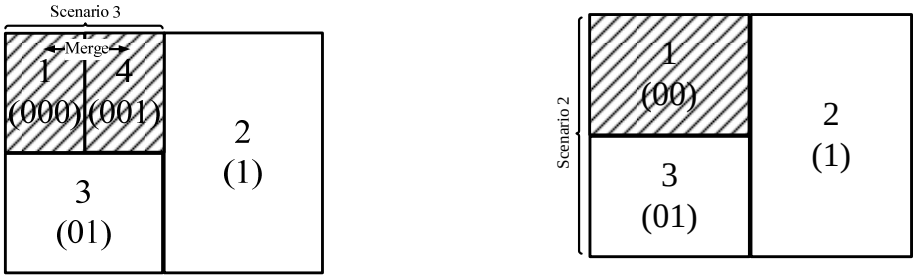


(a) Before merging (b) After merging

Figure 4.16 Scenario 2 to scenario 3

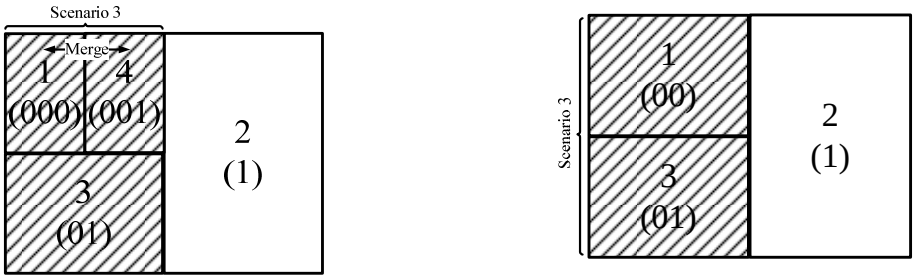
Algorithm for scenario 3

In this scenario, CANS has no valid mergeable-zone. All mergeable-zone-pairs have crashed. We must find the crashed mergeable-zone-pair and merge them. After merging, we have a new bigger crashed zone. Although no crash zone is recovered in this step, CANS may change into scenario 2 or 3 (see Figure 4.17 and Figure 4.18). If CANS changes into scenario 2, one crashed zone will be recovered in the next step.



(a) Before merging (b) After merging

Figure 4.17 Scenario 3 to scenario 2



(a) Before merging (b) After merging

Figure 4.18 Scenario 3 to scenario 3

By combining the above three algorithms, additional insight can be deduced. Every merging recovers one crashed zone except in scenario 3. However, scenario 3 changes into scenario 2 via merging. Table 4.1 and Figure 4.19 show what happens after merging. During merging, the system's scenario keeps changing until all crashed zones are recovered.

Current scenario	Next scenario	Crashed zone recovered
1	1/ 2/ 3	Yes
2	1/ 2/ 3	Yes
3	2/ 3	No

Table 4.1 Transformation between scenarios

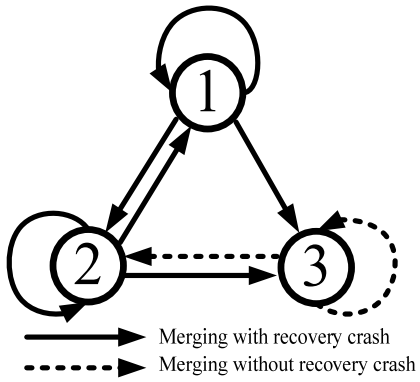


Figure 4.19 Transformation between scenarios

If peers can retrieve any replications, CANS can recover from any multiple crashes as long as one peer survives. However, the replications store in peers (see section 2.4.5). When a peer crashed, the replications on it will crash simultaneously. Therefore, anti-crash capability also depends on replication solution.

4.6 Reliability of the algorithm

In general, as long as one zone survives, the system can recover from multiple crashes. The surviving zone takes over all responsibility. However, CANS must be acceptable at any time. This means that zones cannot arbitrarily merge. Only a mergeable-zone-pair can merge with each other. In this section, we will prove that CANS can recover from any crash. In other words, even if only mergeable-zone-pair merging is allowed, all crashes can be recovered.

We assume that there are enough redundant replicas. All zones can get replicas of crashed zones any time (In this thesis we use the replica solution in [6]). First, we prove that our multiple-crash recovery algorithm can guarantee that CANS always remains acceptable. In our multiple-crash recovery algorithm, we discussed three scenarios in the last section. In the three scenarios, there are only three kinds of merging: valid-zone merging (see Figure 4.10(a)), valid-crashed-zone merging (see

Figure 4.10(b)) and crashed-zone merging (see Figure 4.10(c)). They are all mergeable-zone-pair merging. Hence, CANS always remains acceptable.

Second, we must prove that CANS can recover from any crash via our multiple-crash recovery algorithm. Different kinds of merging have different results, as follows:

- Valid-zone merging: Two valid zones merge with each other. The new zone is valid.
- Valid-crashed-zone merging: A valid zone merges with a crashed zone. The valid zone extends its responsibility to take over the crashed zone. Thus, the new zone is still valid.
- Crashed-zone merging: Two crashed zones merge with each other. The new zone is crashed.

If we let valid zone denote “1” (“true”) and crashed zone denote “0” (“false”), we get Table 4.2. Table 4.2 is the same as the truth table of disjunction. Via calculation, we can determine the situation after multiple mergings.

	zone 1	**zone 2**	**new zone**
Valid-merging	1	1	1
Valid-crash-merging	1	0	1
Valid-crash-merging	0	1	1
Crash-merging	0	0	0
valid: 1, crashed: 0			

Table 4.2 Merging results

Let z_i denote the situation of zone i with $z_i \in \{0,1\}$ and $i \in \{1,2\cdots n\}$.

Let $S_1 = \overbrace{\{z_1, z_2, \cdots\}}^{n}$ denote the situation of CANS before merging, and there is at least one valid peer ($\exists z : z \in S_1 \wedge z = 1$).

Let $S_2 = \overbrace{\{\cdots\}}^{n-1}$ denote situation of CANS after mergings.

According to Table 4.2, a crashed zone will disappear after once merging, thus

$S_2 = S_1 \setminus z \in \{e \mid e \in S_1 \wedge e = 0\}$.

Subsequently,

$S_{i+1} = S_i \setminus z \in \{e \mid e \in S_i \wedge e = 0\}$.

Let S_{end} denote situation of CANS after multiple-crash recovery.

The merging process will end up with $\forall z : z \in S_{end} \wedge z = 1$. All zones in S_{end} are valid. CANS has recovered from multiple-crash.

Subsequently, $\exists z : z \in S_1 \wedge z = 1 \Rightarrow \forall z : z \in S_{end} \wedge z = 1$

We have shown that the algorithm converges and gains a situation S_{end} where all $z \in S_{end}$ are "1," i.e., they are valid. Thus, CANS reaches an acceptable state after finite steps.

4.7 Conclusion

In Chapter 3, CANS uses the CAN tree to reallocate zones, such that they are not concave and slim. It shows that a simple and straightforward solution can achieve the desired zone splitting, but the CAN tree needs to be updated after peer churn. Thus, the system uses extra communication to modify the CAN tree

In order to overcome the drawbacks of the CAN tree, we introduced the zone code and mergeable-zone searching algorithm. They have a similar efficiency and lower communication cost.

Chapter 5

CAN Tree Routing

We talked about routing improvement in section 1.1.2. We need a novel long link routing solution which could improve routing efficiency in low dimensional CAN.

5.1 Introduction

Long links have been extensively utilized by many other Peer-to-Peer protocols, such as Chord [4] and Pastry [5], to improve routing performance. Moreover, we use eCAN [64], LDP [65], SCAN [70], and RCAN [66], which have also adopted long links for the same purpose of improving routing functionality, but in different ways [66]. They are built on top of the conventional CAN overlay.

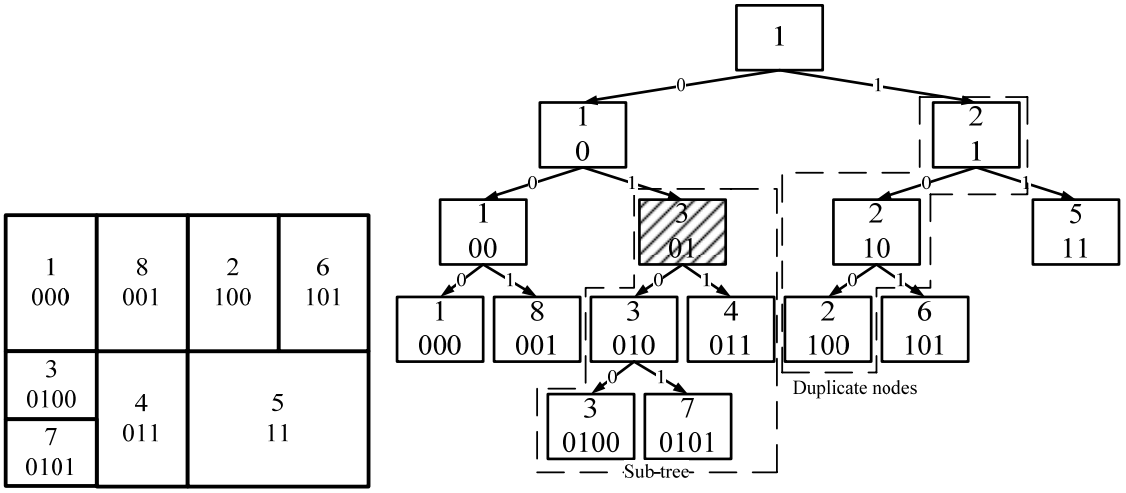
Our scheme is also based on long links. However, we concentrate on a novel approach to establishing a search tree infrastructure in CAN in order to improve its routing performance and enhance fault-tolerance. Meanwhile, both long links and peer churn maintenance overhead should be minimized.

5.2 Zone Code and CAN Tree

Instead of greedy routing, we route in a tree network. The key idea is to establish a Peer-to-Peer tree (CAN tree)[68] via long links. Each peer of CAN is a node of the CAN tree. Because CAN tree is not a binary tree, peers do not know what is next hop. They need more information to choose the routing target peer. We use the zone code.

We proposed the zone code in chapter 4. It is a binary string that records the splitting history of its corresponding zone. In theory, we can obtain the zone code by traversing the partition tree (see Figure 5.1(b)). The partition tree is a binary tree that records the

reassignment process. In order to obtain a zone code, we perform a traversal from root to leaf in the partition tree. This is analogous to Huffman code [69]. Going left is a "0," going right is a "1." A zone code is only completed when a leaf node is reached [62]. Figure 5.1(b) illustrates how zone codes are established via the partition tree.



(a) CAN (b) Partition tree

Figure 5.1 CAN and partition tree

In practice, we do not need the partition tree to generate a zone code. When a peer *p* shares its half zone with a new peer *c*, peer c copies *p*'s zone code. Peer *p* and *c* then append "0" and "1," respectively. Let δ^p denote the zone code of peer *p*. After peer *c* joining, the new zone code of peer *p* is $(\delta^p,0)$ and the zone code of *c* is $(\delta^p,1)$. Hence, zone code grows simultaneously with zone splitting. The more splits, the longer the zone code becomes[62]. Combining all our insights, we deduced the following corollary:

Corollary 5.1: In a partition tree, the zone code of peer *p* is the prefix of the zone codes of all peers in the subtree rooted at peer *p*.

For example, the shaded peer in Figure 5.1(b) has zone code (0, 1). Thus, the zone codes of peers in the subtree have a common prefix (0, 1).

By Corollary 5.1, we can route in the partition tree. Because the internal peers in the partition tree no longer exist (they existed at some previous time), we cannot establish a distributed partition tree via long links in practice. We need the CAN tree to realize the long links.

CAN tree is a variation of the partition tree. Both of them are representations of the zone splitting process. There are some duplicate peers that have the same name but different zone codes in the partition tree (see Figure 5.1(b)). If we merge duplicate peers into one peer and it inherits duplicate peers' children, it becomes a CAN tree (see Figure 5.2). A CAN tree is not a binary tree, but all peers exist in the CAN tree. Thus, we can implement highly efficient routing in the CAN tree.

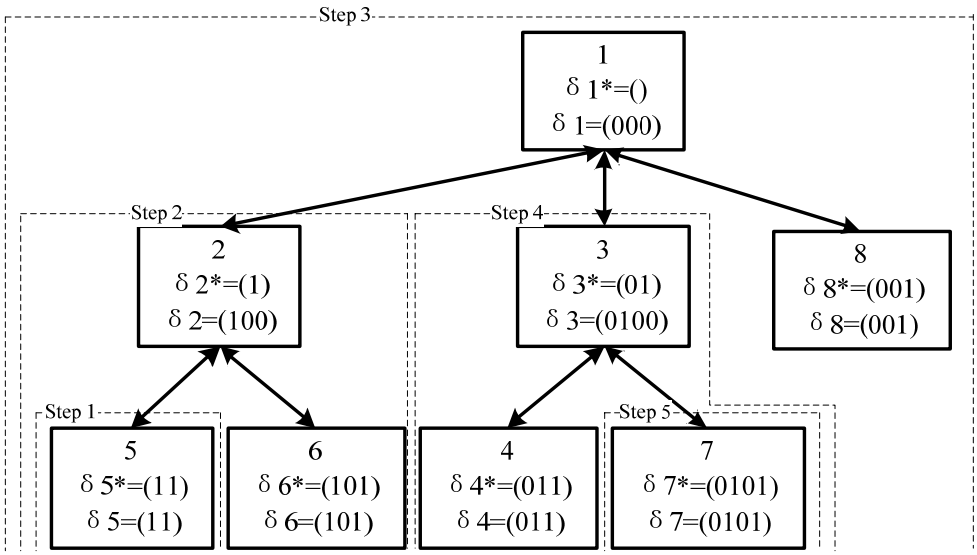


Figure 5.2 CAN tree

We build the CAN tree via parent-child long links. When a new peer *c* forwards a JOIN request and peer *p* shares half its zone with peer *c*, peer *c* becomes the child of peer *p*. All "parent-child" relations constitute a distributed CAN tree. In order to route, each peer must store its original zone code δ^* and current zone code δ. Therefore, when a new peer *c* joins in CAN and obtains its zone from peer *p*, peers *p* and *c* must act as follows (see Figure 5.3):

1. Peer *p* splits its allocated zone in half, retaining half and handing the other half to peer *c*.
2. Peer *p* becomes the parent of peer *c*. Both of them augment long links to establish a "parent-child" relation in the CAN tree.
3. Peer *c* copies *p*'s current zone code ($\delta^p = \delta$). Peers *p* and *c* then append "0" and "1," respectively, i.e., new $\delta^p = (\delta,0)$ and $\delta^c = (\delta,1)$.
4. Peer *c* sets $\delta^{c*} = (\delta,1)$ (original zone code *c*), Peer *p* is not a new peer, therefore δ^{p*} (original zone code *p*) does not change.

(a) Before joining (b) After joining

Figure 5.3 New peer c joins CAN

Let δ^{p*} denote the original zone code of peer *p*. δ^{p*} is the first zone code of peer p, and δ^{p*} is constant. If peer p shares its zone with a new peer, $\delta^{p*} \neq \delta^p$. For example, in Figure 5.2, $\delta^3 = (0,1,0,0)$ and $\delta^{3*} = (0,1)$. δ^{p*} is the prefix of δ^p. Consequently, δ^{p*} is also the prefix of the zone code of the children of peer *p*. Combining all our insights, we deduced the following corollary:

Corollary 5.2: In a CAN tree, peer *p* has the original zone code δ^{p*}. δ^{p*} is the prefix of the zone codes of the peers in the subtree rooted at peer *p*.

For example, in Figure 5.2, δ^{3*} is the prefix of the zone code of all the peers in the subtree rooted at peer 3. δ^{1*} is null, it is the prefix of any zone code of peers in the

CAN tree. Since a new peer obtains its zone code via copying and extending the zone code of its parent, we deduce the following corollary:

Corollary 5.3: If δ^{p*} of peer *p* is the prefix of the δ^{c} of peer *c*, peer *c* is in the subtree rooted at peer *p*.

Let δ^{p*} denote the original zone code of the current peer *p* and δ^{d} the zone code of the target peer *d*. Consequently, our routing scheme is that peer *p* checks whether its δ^{p*} is the prefix of δ^{d}. If it is, peer *p* forwards the message to its child, which shares the longest common prefix with δ^{d}. If not, peer *d* is not in the subtree rooted at peer *p* and so peer *p* forwards the message to its parent peer.

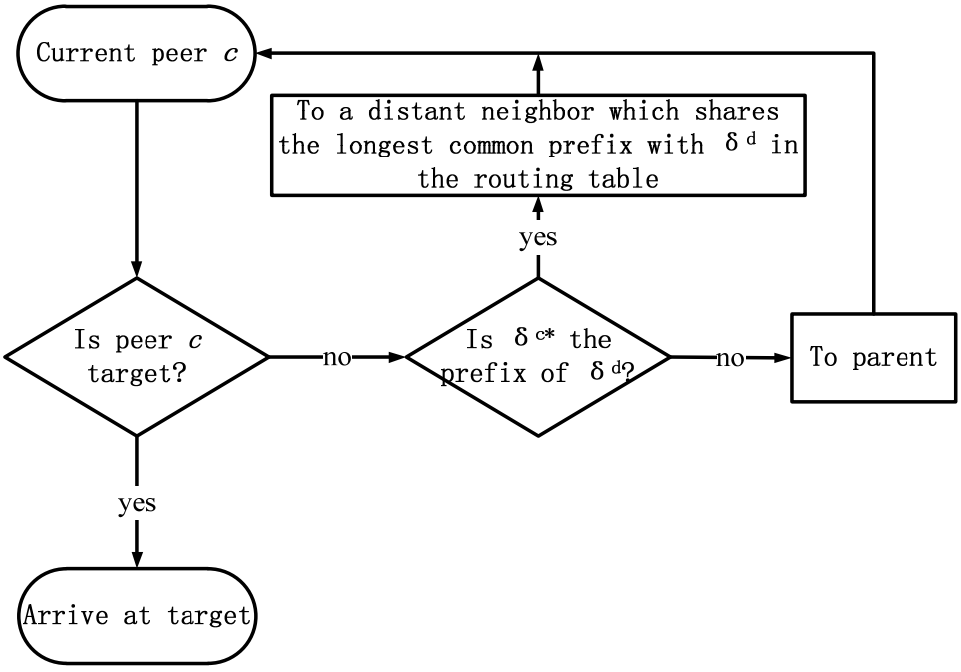


Figure 5.4 Flow diagram

Figure 5.2 illustrates the routing from peer 5 to peer 7. If the target peer is not in the subtree rooted at the current peer, we expand the searching subtree until it covers the target peer. Afterwards, we shrink the searching subtree until the current peer is the

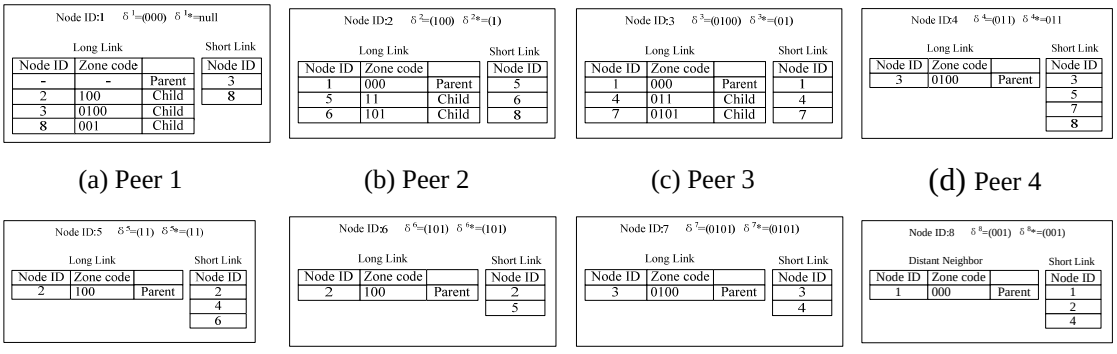
target. During shrinking, the target peer is always in the subtree rooted at the current peer. Therefore, the routing must eventually terminate successfully.

5.3 Routing table

The routing table consists of the short links toward the neighbors and the long links toward the parent and child peers in the CAN tree, and the original zone code δ^* (see Figure 5.5). In this section, we present the details of how to establish and maintain the routing table. The routing procedure is addressed in the next section.

CAN maintains short links by exchanging heartbeat messages between immediate neighbors. For d-dimensional CAN, a peer maintains $O(d)$ neighbors on average. This is analogous to the original CAN.

Long links are a part of the CAN tree; this is central to our scheme. They are established during the joining of new peers. In the beginning, there is only one peer in CAN. This peer is the root of the CAN tree. When a new peer joins in CAN, an existing peer splits its zone into two sub-zones, retaining one and handing the other to the new peer. The two peers are parent and child in the CAN tree. Meanwhile, we establish long links between them, i.e., they augment a long link set in its routing table. They are distant neighbors. The entry in the routing table comprises distant neighbor information, e.g., peer ID, IP address, and zone code δ (Figure 5.5) [71].



(a) Peer 1 (b) Peer 2 (c) Peer 3 (d) Peer 4

(e) Peer 5 (f) Peer 6 (j) Peer 7 (h) Peer 8

Figure 5.5 Routing tables

5.4 Routing Mechanism

If a peer has the zone code δ^d of a target peer, it can choose precisely the next hop via its routing table. Otherwise, we need the algorithm described in 5.4.3 to compute the zone code δ^d based on a target point coordinate.

5.4.1 Routing to a Peer via CAN Tree

Figure 5.4 illustrates the routing flow diagram. If current peer c is not the target peer, it checks whether its original zone code δ^{c*} is a prefix of δ^d. If it is, it forwards the message to the distant peer that shares the longest common prefix zone code with δ^d. If not, it forwards the message to its parent peer. Since the root peer has no parent in the CAN tree, it always forwards the message to its child, which shares the longest prefix with δ^d. As expressed in section 5.2, the routing must eventually terminate successfully.

For example, peer 5 ($\delta^5 = (1,1)$ and $\delta^5 = (1,1)$) in Figure 5.2 forwards a message to peer 7 ($\delta^7 = (0,1,0,1)$). The routing table is in Figure 5.5. The routing process is as follows:

1. Peer 5 is not the target. Since $\delta^{5*} = (1,1)$ is not the prefix of $\delta^7 = (0,1,0,1)$, peer 5 forwards the message to its parent peer 2.
2. Peer 2 is not the target. Since $\delta^{2*} = (1)$ is not the prefix of $\delta^7 = (0,1,0,1)$, peer 2 forwards the message to its parent peer 1.
3. Peer 1 is not the target. Since it is root peer, it forwards the message to the child peer 3 whose $\delta^3 = (0,1,0,0)$ shares the longest common prefix zone code with $\delta^7 = (0,1,0,1)$

4. Peer 3 is not the target. Since $\delta^{3*} = (0,1)$ is the prefix of the $\delta^7 = (0,1,0,1)$, peer 3 forwards the message to child peer 7, which is the target. Thus, routing is finished.

If a peer forwards a message to a point in key space, it has no idea about the target peer. In order to route according to our routing table, it needs to calculate the target zone code. We describe how to do this below.

5.4.2 Get Zone Point Set via Zone-Code

By definition, all zones with the same zone code length are the same size. The zone code of peer p ($\delta^p = (c_1^p, c_2^p, c_3^p \cdots)$) is divided into *d* parts. The sub-set of the zone code $\delta_i^p = (c_{j_1}^p, c_{j_2}^p, c_{j_3}^p \cdots)$ ($j_n \bmod d = i$) records the splitting process along the i-th axis [71].

Given that the zones are halved along one dimension during split, this implies that their sizes are also proportional to the inverse of powers of 2. $|\delta_i^p|$ is the length of δ_i^p, and the proportion of p's width to space's width on the i-th dimension is $\frac{1}{2^{|\delta_i^p|}}$.

Let $(\delta_i^p)_{10}$ denote the decimal representation for δ_i^p and w_i denote CAN's key space width on the i-th dimension. Then, $\frac{(\delta_i^p)_{10} \cdot w_i}{2^{|\delta_i^p|}}$ is p's low boundary on the i-th dimension, and $\frac{((\delta_i^p)_{10}+1) \cdot w_i}{2^{|\delta_i^p|}}$ is p's upper boundary on the i-th dimension (see Figure 5.6) [71].

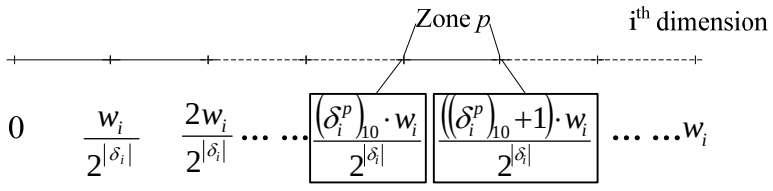


Figure 5.6 Zone boundaries in one dimension

For example, two-dimensional CAN has the width w and height h. The zone code of p is divided into the partial zone codes δ_x^p and δ_y^p, which record the x-axis and y-axis splitting process, respectively. The zones are defined as a set of points $Z_{x,y}^p$:

$$Z_{x,y}^p = \left\{ (x,y) \middle| \left(\delta_x^p\right)_{10} \cdot \frac{w}{2^{|\delta_x^p|}} \le x < (\left(\delta_x^p\right)_{10} + 1) \cdot \frac{w}{2^{|\delta_x^p|}} \quad \left(\delta_y^p\right)_{10} \cdot \frac{h}{2^{|\delta_y^p|}} \le y < (\left(\delta_y^p\right)_{10} + 1) \cdot \frac{h}{2^{|\delta_y^p|}} \right\}$$

Equation 5.1 Zone point set in two-dimensional key space

Therefore, if peer 6 has zone code $\delta^6 = (1,0,1)$ in CAN (w = 800 and h = 600, shown in Figure 5.7), it follows that:

$$\delta^6 = \underset{x\ y\ x}{101} \Rightarrow \begin{cases} \delta_x^6 = 11 \\ \delta_y^6 = 0 \end{cases} \Rightarrow \begin{cases} \left(\delta_x^6\right)_{10} = 3 \\ \left(\delta_y^6\right)_{10} = 0 \end{cases} \text{ and } \begin{cases} |\delta_x^6| = 2 \\ |\delta_y^6| = 1 \end{cases}$$

$$Z_{x,y}^6 = \left\{ (x,y) \middle| \left(\delta_x^6\right)_{10} \cdot \frac{w}{2^{|\delta_x^6|}} \le x < (\left(\delta_x^6\right)_{10} + 1) \cdot \frac{w}{2^{|\delta_x^6|}} \quad \left(\delta_y^6\right)_{10} \cdot \frac{h}{2^{|\delta_y^6|}} \le y < (\left(\delta_y^6\right)_{10} + 1) \cdot \frac{h}{2^{|\delta_y^6|}} \right\}$$

$$Z_{x,y}^6 = \left\{ (x,y) \middle| 3 \cdot \frac{800}{2^2} \le x < (3+1) \cdot \frac{800}{2^2} \quad 0 \cdot \frac{600}{2^1} \le y < (0+1) \cdot \frac{600}{2^1} \right\}$$

$$Z_{x,y}^6 = \{ (x,y) | 600 \le x < 800 \quad 0 \le y < 300 \}$$

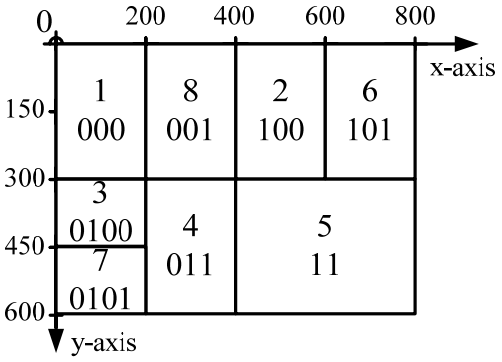


Figure 5.7 CAN (width = 800 and height = 600)

5.4.3 Routing to a Point via Routing Table

The current peer *c* sends a message to a point *p* in the key space, and we assume the target is peer *e* whose zone covers point *p*. In CAN tree routing, we also need the zone

code δ^e of peer *e*. However, the current peer *c* does not have any information about peer *e*. We can calculate the zone code by reversing the aforementioned derivation in section 4.2, and then, forwarding the message to the next peer. Routing is as follows:

1. Calculate δ^e : We calculate δ^e by reversing the aforementioned derivation in section 5.2, e.g., in two-dimensional key space, which is deduced from Equation 5.1. However, Equation 5.1 depends on the length $\left|\delta^e\right|$ of the zone code of peer *e*, which is an unknown factor. We assume that peers *c* and *e* have the same length zone codes, i.e., $\left|\delta^e\right| = \left|\delta^c\right|$. Consequently, δ^e can be deduced from Equation 5.1.
2. Choose next peer: The current peer *c* checks its routing table whether its δ^{c*} is the prefix of δ^e. If it is, it forwards the message to its child peer that shares the longest common prefix with δ^e. If not, it forwards the message to its parent peer. This step is the same as in section 4.1.

For example, peer 5 in Equation 5.1 has $\delta^{5*} = (1,1)$ and $\delta^5 = (1,1)$, and forwards a message to point (100, 500). The routing procedure is as follows:

1. Peer 5 assumes that peer *e* is the target. Since $\left|\delta^5\right| = 2$, set $\left|\delta^e\right| = 2$. Thus, calculate δ^e as follows:

$$Z^e_{x,y} = \left\{ (x,y) \middle| (\delta^e_x)_{10} \cdot \frac{w}{2^{|\delta^e_x|}} \le x < ((\delta^e_x)_{10} + 1) \cdot \frac{w}{2^{|\delta^e_x|}} \quad (\delta^e_y)_{10} \cdot \frac{h}{2^{|\delta^e_y|}} \le y < ((\delta^e_y)_{10} + 1) \cdot \frac{h}{2^{|\delta^e_y|}} \right\}$$

$$\delta^5 = \underset{x\ y}{11} \Rightarrow \begin{cases} \delta^5_x = 1 \\ \delta^5_y = 1 \end{cases} \Rightarrow \begin{cases} \left|\delta^e_x\right| = |\delta^5_x| = 1 \\ \left|\delta^e_y\right| = |\delta^5_y| = 1 \end{cases}, \text{ then}$$

$$\Rightarrow (100,500) \in \left\{ (x,y) \middle| (\delta_x^e)_{10} \cdot \frac{800}{2^1} \le x < ((\delta_x^e)_{10}+1) \cdot \frac{800}{2^1} \quad (\delta_y^e)_{10} \cdot \frac{600}{2^1} \le y < ((\delta_y^e)_{10}+1) \cdot \frac{600}{2^1} \right\}$$

$$\Rightarrow (\delta_x^e)_{10} = 0 \quad and \quad (\delta_y^e)_{10} = 1$$

$$|\delta^5| = 2 \Rightarrow |\delta^e| = 2$$

$$\Rightarrow \begin{cases} \delta_x^e = 0 \\ \delta_y^e = 1 \end{cases}$$

$$\Rightarrow \delta^e = (0,1)$$

Since $\delta^{5*} = (1,1)$ is not the prefix, it forwards the message to its parent peer 2.

2. Peer 2 calculates and obtains $\delta^e = (0,1,0)$ dependent on $|\delta^2| = 3$ (same as the first step). Since $\delta^{2*} = (1)$ is not the prefix of $\delta^e = (0,1,0)$, it forwards the message to its parent peer 1.
3. Peer 1 calculates and obtains $\delta^e = (0,1,0)$ dependent on $|\delta^1| = 3$. Since it is root peer, it forwards the message to child peer 3 whose $\delta^3 = (0,1,0,0)$ shares the longest common prefix with δ^e.
4. Peer 3 calculates and obtains $\delta^e = (0,1,0,1)$ dependent on $|\delta^3| = 4$. Since $\delta^{3*} = (0,1)$ is the prefix of δ^e, it forwards the message to child peer 7, which is the target. Thus, routing is finished.

Calculating the target zone code is the only difference between routing to a peer and a point.

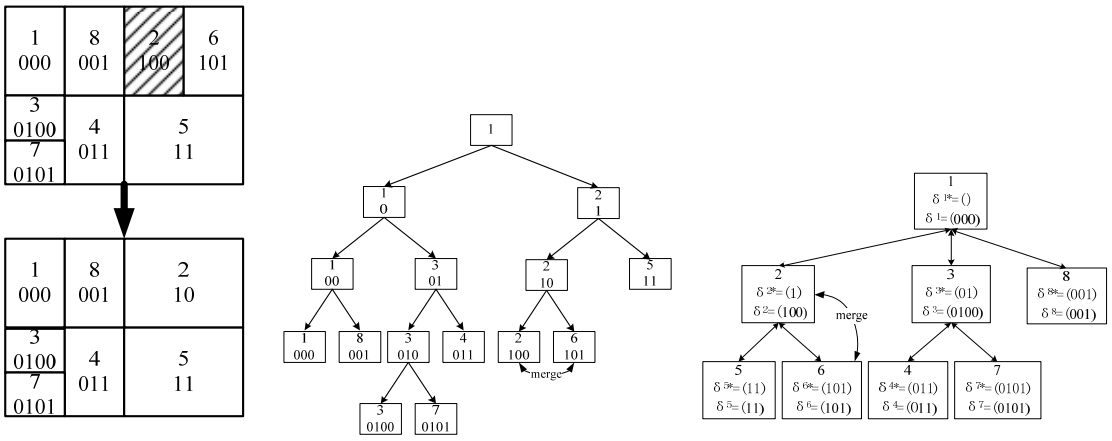
5.4.4 Distant neighbor failure

Peer failure causes potential long link routing failure. If long link routing fails[72], the peer forwards this message in accordance with the original CAN greedy routing. When a peer detects a crash, the recovery procedure is triggered. We describe the process in detail in the next section.

5.5 Peer departure and recovery of CAN Tree

In our scheme, only sibling peers in the partition tree are allowed to merge with each other; they are called mergeable-zone peers, i.e., zones of two peers can merge with each other, which share common zone code prefix and only the last bits are different (peers 2 and 6 in Figure 5.8(b)). In a CAN tree, they are parent and child (see Figure 5.8(c))[68]. When a peer *p* leaves the system and *q* takes over its zone, either of two scenarios may happen:

Case 1 (*p* and *q* are mergeable-zone peers): This case is very straightforward. Peer *q* inherited the long links of *p*, and then peer *q* checks and removes unavailable long links. The zone-code of *q* deletes the last bit of zone code. The original zone code of *q* is the short one between δ^{p*} and δ^{q*}. After a short stabilization period, inform all affected peers to update their routing table. For example, once peer 2 crashes, peer 6 will extend its responsibility to take over the zone of peer 2, and peer 6 modifies its zone code from (1,0,1) to (1,0)(see Figure 5.8(a)). Since $\left|\delta^{2*}\right| < \left|\delta^{6*}\right|$, we set new δ^{6*} to be δ^{2*}.

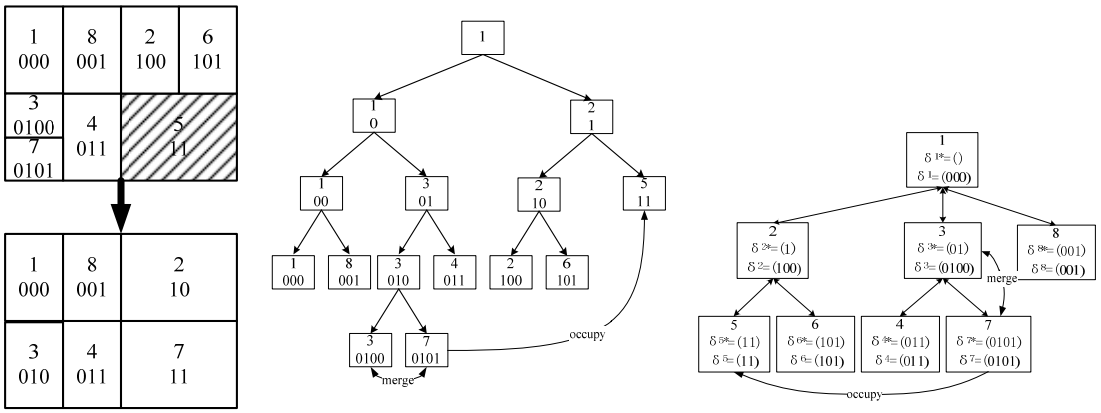


(a) CAN (b) Partition tree (c) CAN tree

Figure 5.8 Merging

Case 2 (*p* and *q* are not mergeable-zone peers): In this case, *q* takes on the role of *p*. Peer *q* also abandons its routing table, and copies *p*'s. The situation turns into the first

case. For example, we need a non-neighbor mergeable-zone peer to deal with peer 5 crash (see Figure 5.9(a)). If we use peer 7 to occupy peer 5, peer 3 will merge with the zone, which is released by peer 7 (see Figure 5.9(b) and Figure 5.9(c)).



(a) CAN (b) Partition tree (c) CAN tree

Figure 5.9 Occupation

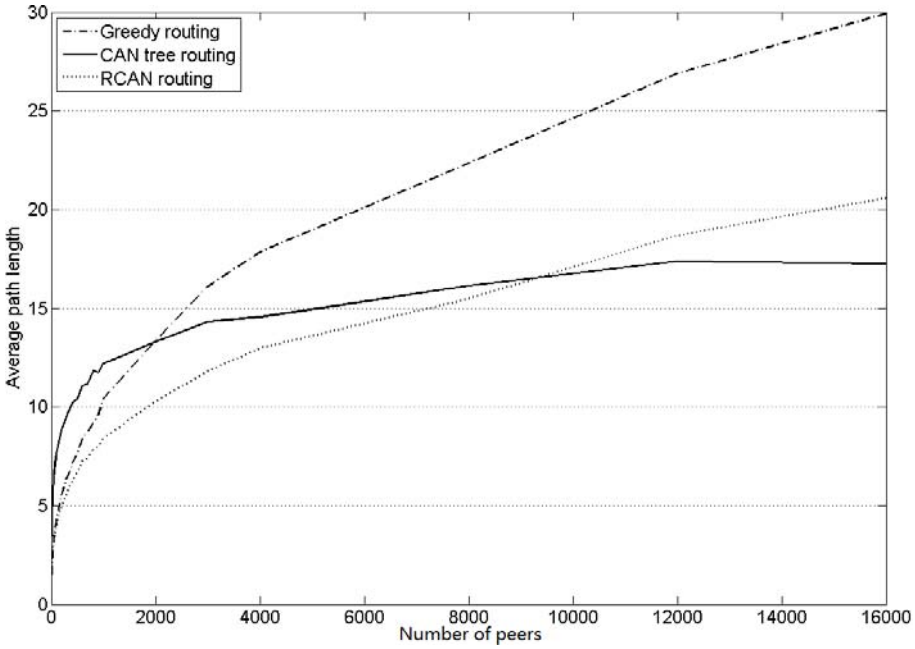
5.6 Evaluation

Our solution does not redesign CAN routing, but extends it. Via long links, several routing features are optimized simultaneously: small routing path, more routing flexibility, and fault-tolerance. The routing procedure always converges, since each step forwards the message to a peer that shares a longer prefix than the last step, each step moves closer to the target.

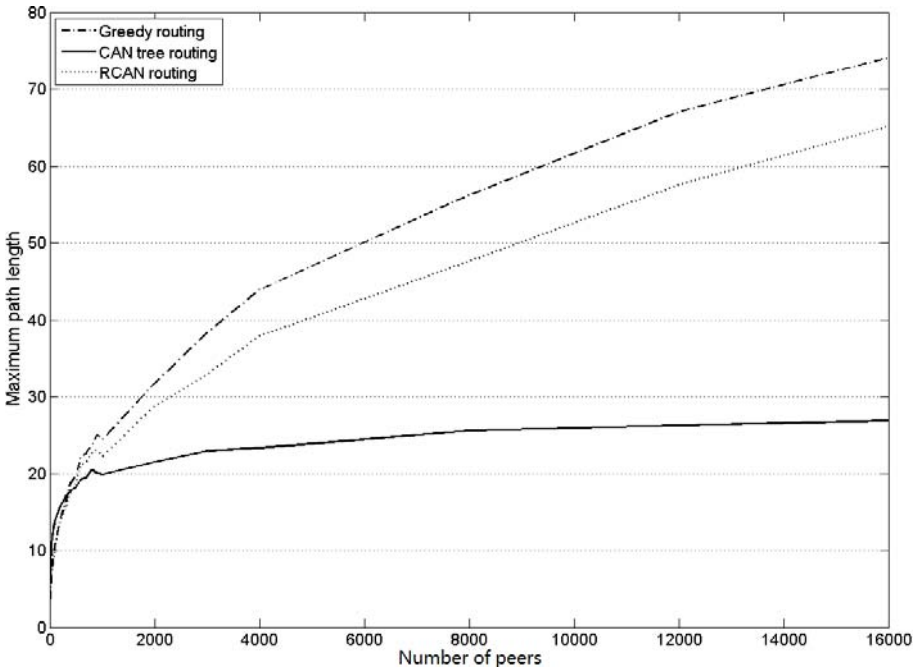
CAN tree routing is based on a tree. Hence, the complexity depends on the tree structure. In order to demonstrate the effectiveness of our design in terms of routing performance, we implemented a CAN tree routing scheme in C# and conducted a set of experiments via distinct schemes on networks with up to 16000 peers. We ran CAN tree routing against the original CAN greedy routing to offer comparative measurements. These measures include essentially: path length to cope with different network size, path length distribution, and number of long links per peer.

Figure 5.10(a) and Figure 5.10(b) show plots of the average and the maximum path length, respectively, with respect to network size. The path length is measured by the

number of hops traversed during each lookup request. Figure 5.10 illustrates that both the average and maximum path length in CAN tree routing are better than other routing, and both of them are perfectly asymptotic to the logarithm of the peers. The path length of greedy routing (see Figure 5.10) increases much faster.



(a) Average path length



(b) Maximum path length

Figure 5.10 Path length with increasing network size

Figure 5.11 illustrates the path lengths distribution of routing in CAN with 16000 peers. The path length distribution of CAN tree routing is much better than in other routing.

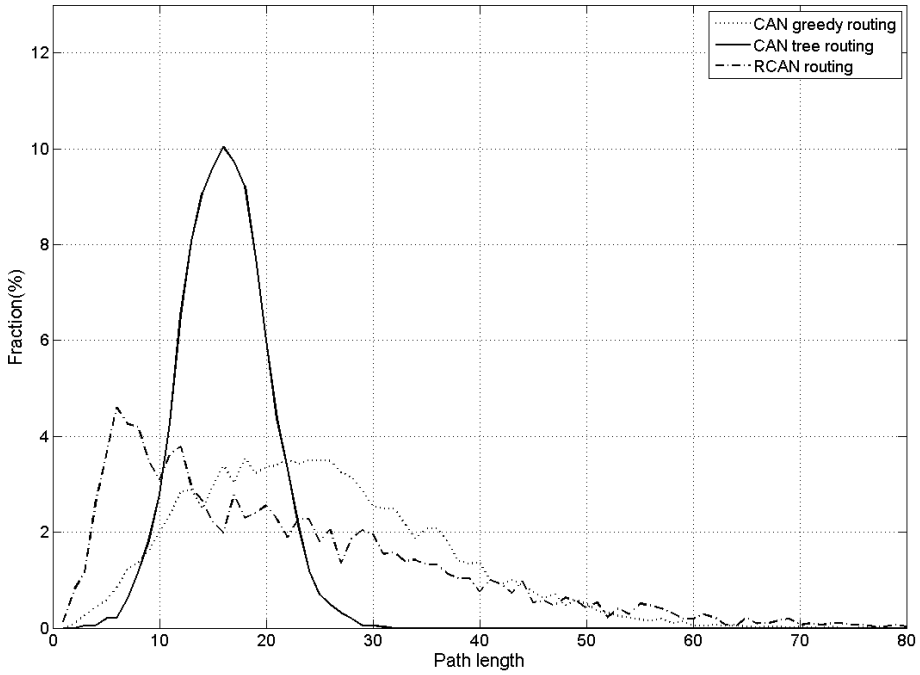


Figure 5.11 Path length distribution

Except for the first peer, every new peer needs two long links to join in the CAN tree. Each parent peer needs one long link pointing to its child; and child peers need one long link pointing to its parent. Hence, the number of long links is $(n-1)\times 2$, and the average long links can be calculated as follows:

$$L_{average} = \frac{(n-1)\times 2}{n}$$

$$\lim_{n \to \infty} L_{average} = \lim_{n \to \infty} \frac{(n-1)\times 2}{n} = 2$$

Thus, each peer maintains two long links on average.

5.7 Conclusion

CAN tree routing is a novel routing scheme based on the CAN tree to overcome the weakness of greedy routing in CAN. CAN with CAN tree routing is a completely

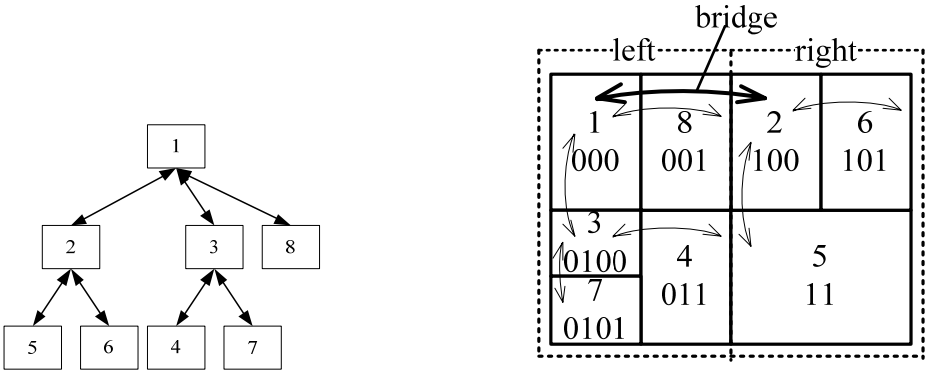
decentralized system. The CAN tree infrastructure gracefully adapts itself to cope with any changes in the network. As a pure Peer-to-Peer system, peers assume equal responsibility. The system maintains peers' routing states while minimizing cost even in the presence of high rate of churn. The critical contribution is the equipping of each peer with long links that significantly enhance routing efficiency. Every peer connected with its parent and children. The number of long links per peer is independent of the network size (dependent on the number of children). Thus, the system can scale by several orders of magnitude without loss of efficiency.

Our routing scheme has more links than the original CAN, which incurs a tiny overhead to maintain long links. It has been proved that the number of long links per peer is two on average. However, it has also been shown that the small extension leads to significant improvements in routing performance.

Chapter 6

Zone Code Routing

We proposed CAN tree routing in chapter 5. A CAN tree is a tree structure. The routing performance can easily be boosted from $O(n^{1/d})$ to $O(\log n)$ by equipping each peer with two long links on average. The small extension leads to significant improvements in routing performance. However, the tree infrastructure causes unfair overhead. For example, Figure 6.1 illustrates that a long link between peers 1 and 2 is a bridge that connects left and right peers (Figure 6.1(b)). They are bottlenecks and cause imbalanced routing overhead. Then we proposed zone code routing.



(a) CAN tree (b) CAN and its long links

Figure 6.1 CAN tree routing

The key idea of zone code routing is zone code match. The message is forwarded to a peer which has more common prefix with target. For example in Figure 6.2, A has no common prefix with target T. After first hop, B has 1 bit common prefix with T. After second hop, C has 2 bits common prefix with T. The length of zone code is limited. Therefore, the message could arrive at target anyhow.

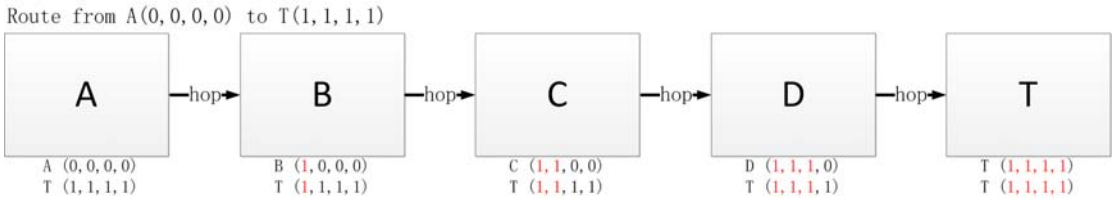


Figure 6.2 Zone code routing

In order to realize zone code routing, every peer needs long links to distant neighbors which have special zone codes.

6.1 Routing Table

In order to route, each peer in CAN maintains a routing table, whose entries include information about its adjacent peers' network address, distant neighbor's network address, zone code, etc. Using the routing table, the current peer can directly communicate with its immediate/distant neighbors. In this section, we present the details of how a routing table is established and maintained. The routing mechanism is addressed in the next section.

CAN maintains short links by exchanging heartbeat messages between immediate neighboring peers. For d-dimensional CAN, a peer maintains $O(d)$ neighbors on average. This is analogous to original CAN.

In order to enhance hop span, peers need long links. The long links connected with some distant neighbors which have special zone codes. First distant neighbor zone code has no common prefix with current peer. Second distant neighbor zone code has 1-bit common prefix, and so on. A peer with *k* bits zone code has *k* long links. The *k*th long link forwards to distant neighbor which has *k*-1 common prefix.

A peer *p* with *k* bits zone code partitions the key space into *k* sub-regions. The size of the sub-region ranges between the size of *p*'s zone and half the key space and they cover the entire key space (see Figure 6.3).

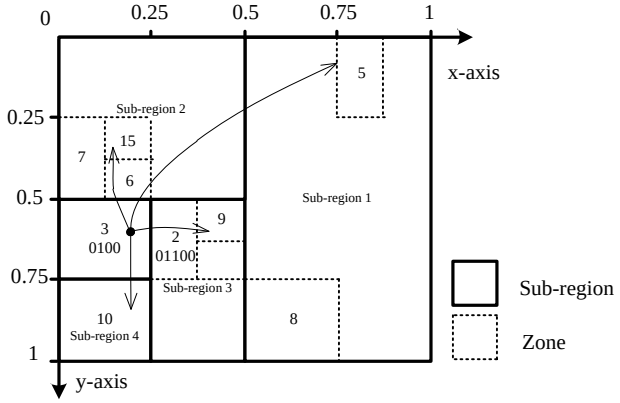


Figure 6.3 Sub-regions and long links for peer 3

6.1.1 Sub-regions

Since the zone code might define the boundaries of the zone, it can also define the boundaries of the sub-region. The sub-regions of peer p ($\delta^p = \left(c_1^p, c_2^p, c_3^p \cdots, c_k^p\right)$) are represented as follows:

- sub-region 1: $\delta^p_{sub-region1} = \left(\overline{c_1^p}\right)$
- sub-region 2: $\delta^p_{sub-region2} = \left(c_1^p, \overline{c_2^p}\right)$
- sub-region 3: $\delta^p_{sub-region3} = \left(c_1^p, c_2^p, \overline{c_3^p}\right)$

  …
- sub-region k: $\delta^p_{sub-region\_k} = \left(c_1^p, c_2^p, c_3^p \cdots, \overline{c_k^p}\right)$

The zone code of the sub-region is deduced as follow:

$$\delta^p_{sub-region\_n} = \left(c_1^p, c_2^p, \cdots, \overline{c_n^p}\right)$$

Equation 6.1 Zone code of sub-region

Consequently, the sub-regions for peer 3 ($\delta^3 = (0,1,0,0)$) in Figure 6.3 are defined as follows:

1. sub-region 1: $\delta^3_{sub-region1} = \left(\overline{c_1^3}\right) = (\overline{0}) = (1)$, then

$$\text{Equation 5.1} \Rightarrow Z^3_{sub-region1} = \left\{(x,y) \middle| (0.5 \le x < 1 \quad 0 \le y < 1\right\}$$

2. sub-region 2: $\delta^{\,3}_{sub-region\ 2} = \left(c_1^{\,p}, \overline{c_2^{\,p}}\right) = (0,\overline{1}) = (0,0)$, then

$$\text{Equation 5.1} \Rightarrow Z^3_{sub-region2} = \left\{(x,y) \middle| (0 \le x < 0.5 \quad 0 \le y < 0.5\right\}$$

3. sub-region 3: $\delta^3_{sub-region3} = \left(c_1^3, c_2^3, \overline{c_3^3}\right) = (0,1,\overline{0}) = (0,1,1)$, then

$$\text{Equation 5.1} \Rightarrow Z^3_{sub-region3} = \left\{(x,y) \middle| (0.25 \le x < 0.5 \quad 0.5 \le y < 1\right\}$$

4. sub-region 4: $\delta^3_{sub-region4} = \left(c_1^3, c_2^3, c_3^3, \overline{c_4^3}\right) = (0,1,0,\overline{0}) = (0,1,0,1)$, then

$$\text{Equation 5.1} \Rightarrow Z^3_{sub-region4} = \left\{(x,y) \middle| (0 \le x < 0.25 \quad 0.75 \le y < 1\right\}$$

A sub-region consists of one or more zones. Zone code is prefix code. If a zone is a resident of a sub-region, the sub-region's zone code is the prefix of the zone's zone code. For example, zone 2 is a resident of sub-region 3. $\delta^3_{sub-region3} = (0,1,1)$ is the prefix of $\delta^2 = (0,1,1,0,0)$.Combining all our insights, we deduced the following corollary:

Corollary 6.1: If peer *p* and peer *s* are located in the same sub-region and peer *d* is out of the sub-region, the common zone code prefix between peers *p* and *s* is inevitably longer than the common prefix between peers *p* and *d*.

In other words, peers in sub-region 1 have no common prefix with current peer. Peers in sub-region 2 have 1-bit common prefix, and so on. A peer with *k* bits zone code has *k* sub-regions. Peers in sub-region *k* has *k*-1 common prefix.

6.1.2 Establishing the Long Links

Peer *p* selects a random point from each sub-region, and then it routes DISCOVER messages to the random points in key space. The corresponding peers, of which the zones cover those points, will be distant neighbors of peer *p*. In the aforementioned example, peer 3 has the distant neighbors listed in Table 6.1.

Sub-region	Random point	Distant neighbor
$Z^3_{sub-region1}$	(0.87,0.13)	Peer 5
$Z^3_{sub-region2}$	(0.13,0.32)	Peer 15
$Z^3_{sub-region3}$	(0.38,0.55)	Peer 9
$Z^3_{sub-region4}$	(0.13,0.84)	Peer 10

Table 6.1 Distant neighbors

Hence, the routing table for peer 3 (see Figure 6.4) contains four long links toward its distant neighbors.

Node ID: 3 Zone code:0100

Short Link

Node ID	Zone code
7	00010
6	000111
2	01100
10	0101

Long Link

Node ID	Zone code	Sub-region
5	10100	$Z^3_{sub-region\ 1} = \{(x,y) \mid 0.5 \le x < 1 \quad 0 \le y < 1\}$
15	000110	$Z^3_{sub-region\ 2} = \{(x,y) \mid 0 \le x < 0.5 \quad 0 \le y < 0.5\}$
9	011010	$Z^3_{sub-region\ 3} = \{(x,y) \mid 0.25 \le x < 0.5 \quad 0.5 \le y < 1\}$
10	0101	$Z^3_{sub-region\ 4} = \{(x,y) \mid 0 \le x < 0.25 \quad 0.75 \le y < 1\}$

Figure 6.4 Routing table for peer 3

6.2 Routing Mechanism

In this section, we describe a routing scheme that relies on the routing table mentioned above.

6.2.1 Forward a Message to a Peer

When the current peer knows the zone code of the target peer, it checks its immediate/distant neighbor first. If the target is its neighbor, the peer can directly

forward the message to the target peer. Otherwise, it selects the next hop dependent on the zone code of its neighbors. It then forwards the message to its distant neighbor, for which the zone code shares the longest common prefix with the zone code of the target peer. For example, peer 3 forwards a message to peer 8 ($\delta^8 = (1,1,0,1)$) in Figure 6.3. In the routing table for peer 3 (see Figure 6.4), peer 5's zone code $\delta^5 = (1,0,1,0,0)$ shares 1-bit common prefix with $\delta^8 = (1,1,0,1)$. The other distant neighbors share 0-bit common prefix. Thus, peer 3 forwards the message to peer 5. The target peer 8 is the distant neighbor of peer 5 (see Figure 6.5). The routing is finished.

Node ID: 5 Zone code:10100

Short Link

Node ID	Zone code
12	10001
14	10101
19	101101

Long Link

Node ID	Zone code	Sub-region
2	01101	$Z^5_{sub-region1} = \{(x,y) \mid (0 \le x < 0.5 \quad 0 \le y < 1\}$
8	1101	$Z^5_{sub-region2} = \{(x,y) \mid (0.5 \le x < 1 \quad 0.5 \le y < 1\}$
4	10000	$Z^5_{sub-region3} = \{(x,y) \mid (0.5 \le x < 0.75 \quad 0 \le y < 0.5\}$
11	101101	$Z^5_{sub-region4} = \{(x,y) \mid (0.75 \le x < 1 \quad 0.25 \le y < 0.5\}$
14	10101	$Z^5_{sub-region5} = \{(x,y) \mid (0.875 \le x < 1 \quad 0. \le y < 0.25\}$

C

Figure 6.5 Routing table for peer 5

6.2.2 Forward a Message to a Point

When a peer needs to forward a message to a point in the key space and does not have any information about the target peer, the routing procedure is divided into two steps. First, the peer checks whether the target point is covered by one of its immediate/distant neighbors. If this is the case, the peer can directly forward the message to the target peer. The routing then finishes. For example, peer 3 in Figure 6.3 forwards a message to the point (0.82, 0.21). Since the distant neighbor peer 5

takes responsibility for the zone $Z^5_{x,y} = \{(x, y) | (0.75 \le x < 0.875 \quad 0 \le y < 0.25\}$ that covers the point (0.82, 0.21), the message is forwarded to peer 5 via long link.

If the target is not an immediate/distant neighbor, the current peer looks up the corresponding sub-regions, which covers the target point, and forwards the message to the corresponding distant neighbor. For example, peer 3 in Figure 6.3 forwards a message to the target point (0.56, 0.87). Since sub-region 1 covers the target point (0.56, 0.87), it forwards the message to peer 5. Peer 5 has five long links (see Figure 6.6) and its routing table is shown as Figure 6.5. Consequently, peer 5 has a distant neighbor peer 8 whose zone covers the target point (0.56, 0.87). Hence, peer 5 directly forwards the message to peer 8. The routing then finishes.

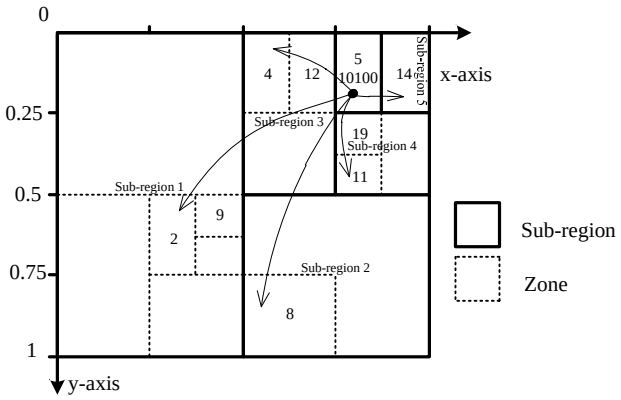


Figure 6.6 Sub-regions and long links for peer 5

In the two aforementioned long-link-routings, the message is forwarded to the distant neighbor located in the same sub-region as the target. Since Corollary 6.1 in Section 6.2.2.1, the next peer shares longer common zone code prefix with the target peer. The length of the zone code is finite. Hence, the routing must eventually terminate successfully.

6.3 Peer Churn

.After new peers joining CAN, they need to establish long links according to section 6.1.2. We have introduced a method of generating the zone code by partition tree in

section 3.2. Although zone codes can be generated by different approaches, the zones are all exactly mapped onto zone code space via their zone codes (see Figure 6.7).

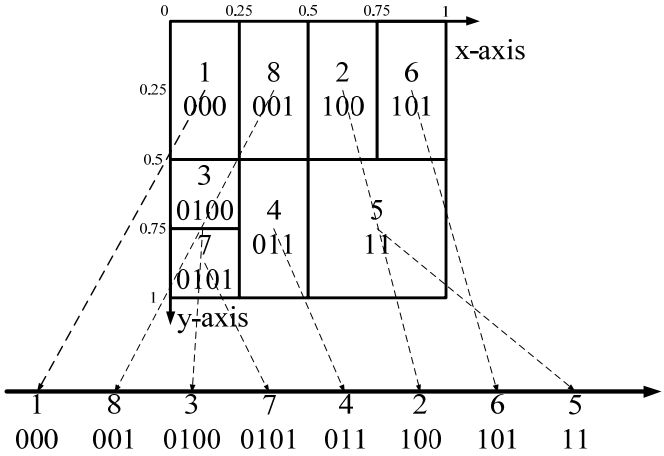


Figure 6.7 CAN (width = 1 and height = 1)

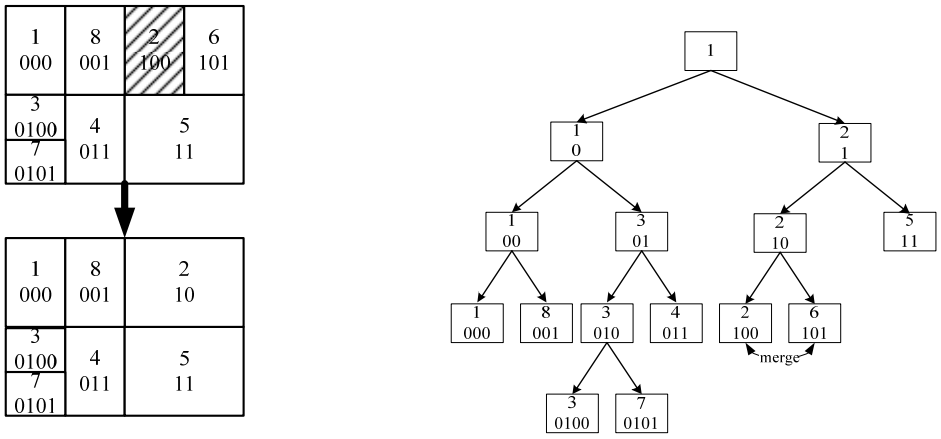
When peers leave or crash, CAN must ensure that all zones are rectangles. In CANS, only sibling peers in a partition tree are allowed to merge with each other (peers 1 and 8, peers 3 and 7, and peers 2 and 6 in Figure 6.8). They are termed mergeable-zone peers[68]. Since the zone code is prefix code (Corollary 4.2 in Section 4.2), we deduced the following corollary:

Corollary 6.2: The mergeable-zone peers share common zone code prefix and only last bits are different.

If a peer actively leaves, it will forward a message to its smallest neighbor. This neighbor is in charge of recovery. Otherwise, the crashed peer (passive leaving) will be detected via heartbeat messages by its neighbors. When a peer finds its neighbor crashed, it takes responsibility of recovery. The difference between active leaving and passive leaving is who will be in charge of recovery. In active leaving the leaving peer has the chance to choose a neighbor to recover the system. In passive leaving the first peer detecting that a neighbor crashed has the responsibility to fix the peer-to-peer structure.

When a peer *p* leaves the system and *q* takes over its zone, either of two scenarios may occur:

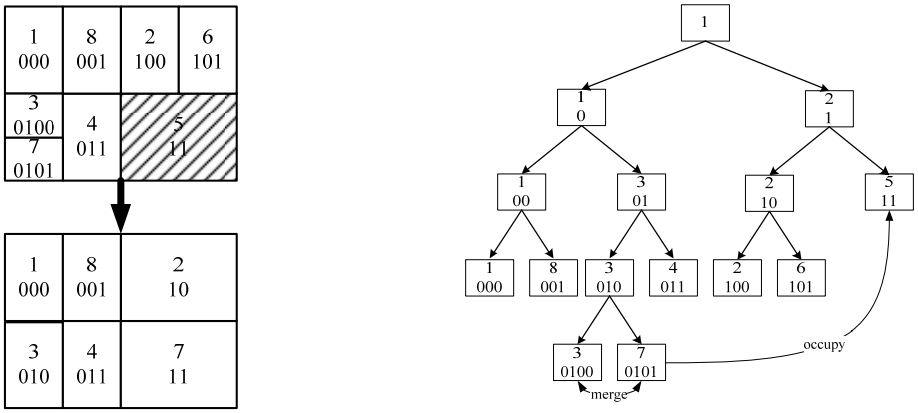
- Case 1 (*p* and *q* are mergeable-zone peers): This case is very straightforward. Peer *q* inherits the long links of *p*, and then peer *q* checks and removes unavailable long links. From Corollary 6.2, the zone code of *q* deletes the last digit of the zone code after merging. After a short stabilization period, it informs all affected peers to update their routing tables. For example, once peer 2 in Figure 6.8 crashes, peer 6 extends its responsibility to take over the zone of peer 2, and peer 6 modifies its zone code from (1,0,1) to (1,0).



(a) CAN (b) Partition tree

Figure 6.8 Merging

- Case 2 (*p* and *q* are not mergeable-zone peers): In this case, *q* takes the role of *p*. Peer *q* abandons its routing table and zone code and copies *p*'s routing table and zone code from *p*'s replication. Consequently, the scenario becomes similar to the first case. For example, we need non-neighbor mergeable-zone peers to deal with peer 5 in the Figure 6.9 crash. If peer 7 occupies peer 5, peer 3 will merge with the zone, which is released by peer 7.



(a) CAN (b) Partition tree

Figure 6.9 Occupation

Peer departures and crashes cause potential long link routing failures. If long link routing fails, the peer forwards this message according to the original CAN greedy routing. When a peer detects a crash, a recovery procedure is triggered. The peer selects a random point in the sub-region, in which the long link failed and forwards DISCOVER messages to the random point. The corresponding peer for which the zone covers those points becomes the new distant neighbor. The broken long links are recovered.

6.4 Evaluation

6.4.1 Reliability

By means of long links, the routing is optimized with more flexibility and fault-tolerance. Since the message is always forwarded to a peer that shares a longer common prefix with the target peer, the routing procedure eventually converges. Zone code routing does not abandon CAN greedy routing, but extends it. After equipping long links, peers have more degrees of freedom and more choices for routing. When a long link fails, a peer can forward the message via greedy routing. Furthermore, a peer might forward a message across crashed peers in the partitioning (see Figure 6.10). In the same scenario, the greedy routing cannot forward the message to the target. Therefore, our scheme has more fault-tolerance and is more reliable.

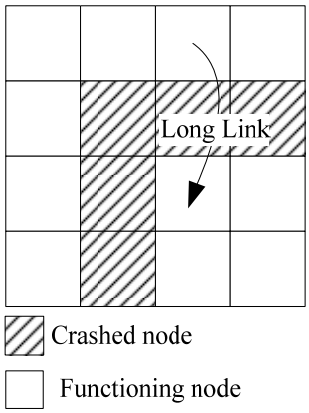


Figure 6.10 Network partition

6.4.2 Routing Evaluation and Cost of Long Link

In our scheme, d-dimensional zones are mapped onto one-dimensional zone code space. If we array all zone-codes of a CANS in ascending order, they are distributed in one-dimensional space which is called zone code space. The zone code realizes the mapping between d-dimensional key space and one-dimensional zone code space.

The zone codes map the d-dimensional key space onto the one-dimensional zone code space. Since zone codes are prefix codes (see Corollary 4.2 in Section 4.2), the order of zone code arrangement in zone code space is the same as the leaf-nodes pre-order traversal (see Figure 6.11). Without the partition tree, we nevertheless have an easy approach to determining the relative positions between two zone codes in zone code space. Let $\left|\delta_{common\_prefix}\right|$ denote the length of the common prefix between them, and the bigger zone code, of which the $\left(\left|\delta_{common\_prefix}\right|+1\right)^{th}$ digit is "1." The other one is smaller. For example, let δ^p denote the zone code of peer *p*. Then, the zone of peer 3 is mapped onto the zone code $\delta^3=(0,1,0,0)$ and the zone of peer 7 is mapped onto the zone code $\delta^7=(0,1,0,1)$. They have the common prefix $\delta^{3|7}_{common\_prefix}=(0,1,0)$ and $\left|\delta^{3|7}_{common\_prefix}\right|=3$. The fourth digitals of δ^7 and δ^3 are "1" and "0," respectively. Thus, δ^7 is bigger and δ^7 is located to the right of δ^3 (see Figure 6.11).

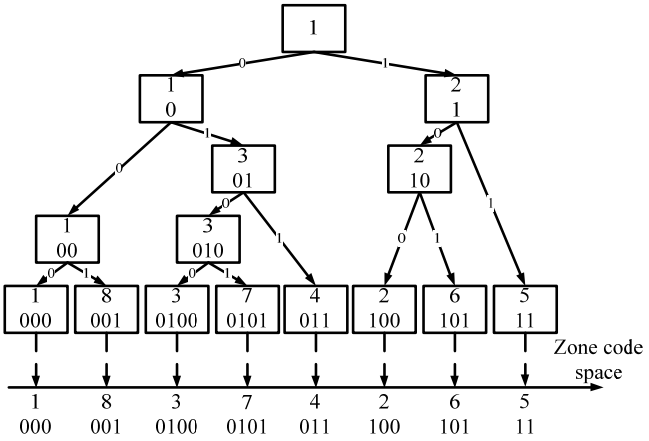


Figure 6.11 Zone code space

When the key space is divided into *k* sub-regions, the zone code space is also divided into *k* intervals. Figure 6.12 illustrates the mapping between two spaces in the aforementioned example. Each sub-region is mapped onto the corresponding interval. Thus, the long links in the zone code space is similar to Chord's finger table.

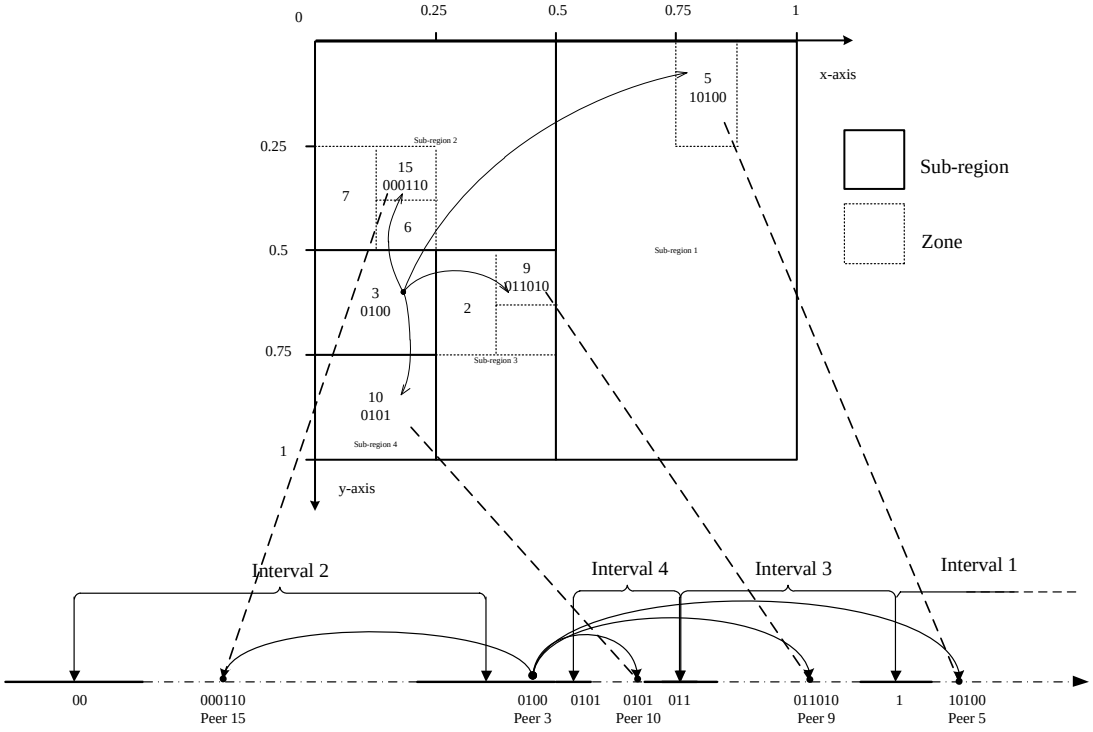


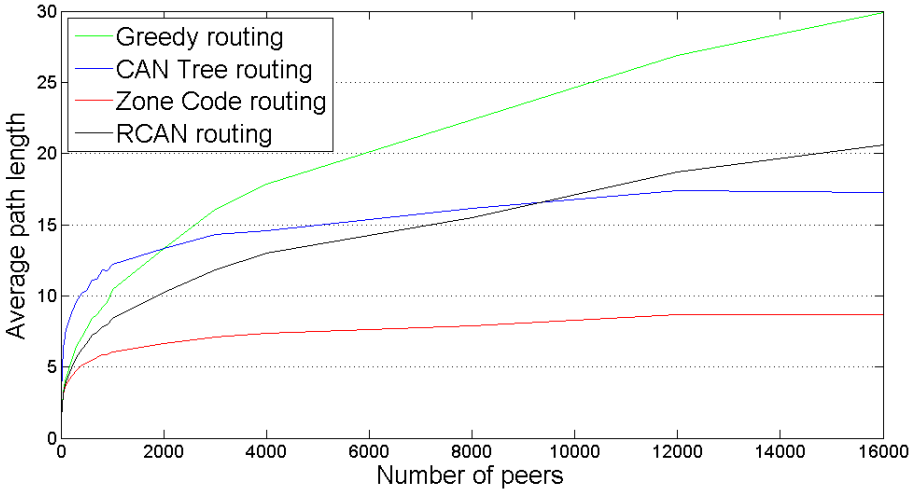
Figure 6.12 Sub-regions in zone code space

Our zone code routing realizes big interval hops in the zone code space to enhance routing efficiency. Given the power-of-two intervals in zone code space, each hop

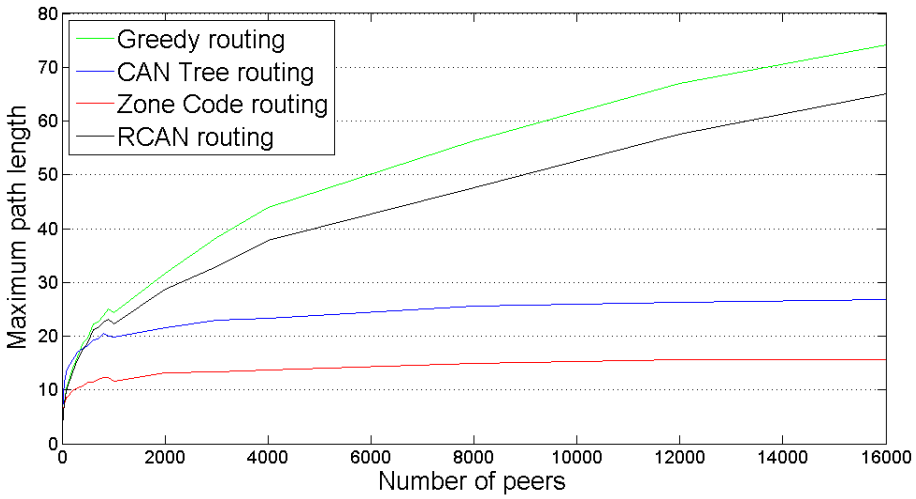
covers at least half of the remaining distance in the zone code space between the current peer and the target peer. This results in an average of $O(\log n)$ routing hops for a CAN with *n* participating peers.

In order to demonstrate the effectiveness of our design in terms of routing performance, we implemented a zone code routing scheme in C# and conducted a set of experiments via distinct schemes on networks with up to 16000 peers. We ran CAN tree routing against the original CAN greedy routing, CAN tree routing, and RCAN routing to offer comparative measurements. These measures included hop count per routing path, and number of long links per peer.

Figure 6.13(a) and Figure 6.13(b) are respective plots of the average and the maximum path length with respect to the network size. The path length is measured in terms of the number of hops traversed during each lookup request. Figure 6.13 illustrates that both the average and maximum path length in zone code routing are better than other routings, and both of them are perfectly asymptotic to the logarithm of peer number. The path length of other routings increases much faster.



(a) Average path length



(b) Maximum path length

Figure 6.13 Path length with increasing network size

Figure 6.14 illustrates the path lengths distribution of routing in CAN with 16000 peers. The path length distribution of zone code routing is much better than others.

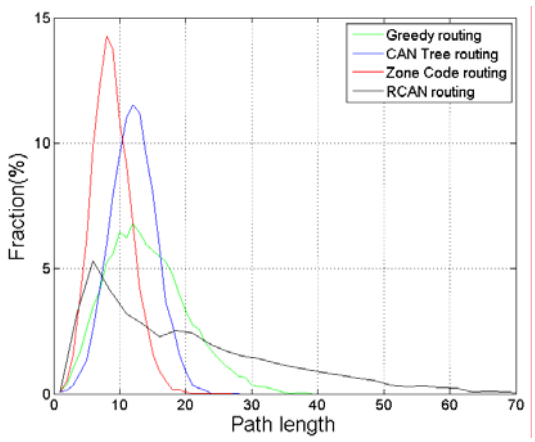


Figure 6.14 Path length distribution

In addition, CAN maintains dynamic long links that scale up to correspond to the network size. Figure 6.15 illustrates that each peer maintains a number of long links on average with respect to the network size. When the CAN has *n* participating peers, each peer maintains $\log_2 n$ long links. The routing state per peer logarithmically scales up to correspond to the number of peers. Hence, the small extension cost leads to significant improvements in routing performance.

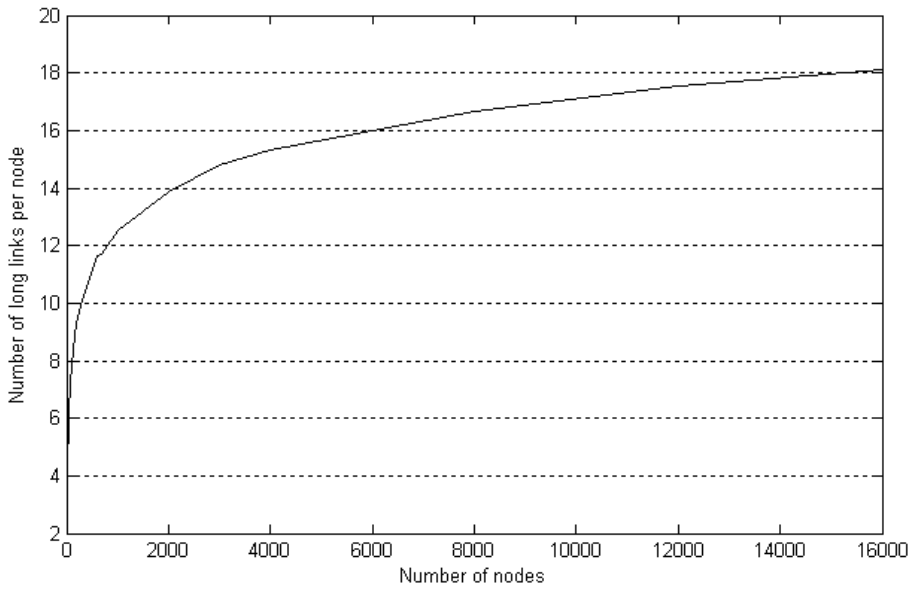


Figure 6.15 Number of long links per peer with increasing network size

6.5 Conclusion

Our zone code routing is a completely decentralized system. It gracefully adapts itself to cope with any changes in the network. As a pure Peer-to-Peer system, peers assume equal responsibility. It overcomes the unfair overhead problem in CAN tree routing.

Using zone code, we map d-dimensional zones onto a one-dimensional zone code space. Zone code routing achieved $O(\log n)$ routing performance with $O(\log_2 n)$ routing state per peer. Thus, the system can scale by several orders of magnitude with high efficiency. Since a zone code is a binary string and it does not need synchronization, the system maintains peers' routing states while minimizing cost even in the presence of a high rate of churn. Consequently, zone code routing significantly improves routing performance while incurring only a small extra overhead.

Chapter 7

Conclusion

With the development of Peer-to-Peer technology, Peer-to-Peer systems are increasingly being utilized in more and more fields. However, new applications must overcome its drawbacks and satisfy new requirements. With respect to the properties of simulation, we faced two challenges in our work—reorganizing of the zone-assignment to solve the “concave and slim problem” and development of novel efficient routing solutions for low-dimensional CAN. Our contributions to the field enable developers to establish their own Peer-to-Peer MMVE/simulation system with a simple structure and high efficiency and maintain the communication overhead as low as possible.

Firstly, we presented a novel approach for reorganizing the zone-assignment in CANS (see Section 3). To achieve this, we used a distributed tree infrastructure (CAN tree) and introduced a search algorithm to lookup mergeable-zone-pairs. CAN tree is highly distributed and thus supplies the required robustness and availability. In addition, we developed a peer churn coping strategy. Thereby, our CAN tree offers resilience against peers leaving and crashing.

Secondly, we proposed zone codes for reorganizing the zone-assignment (see Section 4) of CAN tree lookups mergeable-zones by the tree structure splitting history record. In order to maintain the freshness of CAN trees, peers in a CAN tree need to communicate to modify the tree infrastructure after peers churn. The extra communication increases system overhead. In contrast, the zone code does not need update communication, further reducing the overhead of CANS. Furthermore, we proved its robustness and availability during multiple simultaneous peers failures

We also introduced the CAN tree routing solution (see Section 5). It is designed to efficiently forward messages in low-dimension CAN. The routing performance can easily be boosted from $O(n^{1/d})$ to $O(\log n)$ by equipping each peer with two long links on average. Thereby, a small extension leads to significant improvements in routing performance.

Finally, we presented a zone code routing solution (see Section 6). Since CAN tree routing is based on tree infrastructure, the peers have unfair routing overhead—zone code routing overcomes this drawback. It achieves $O(\log n)$ routing performance with $O(\log_2 n)$ routing state per peer. Hence, the system can scale by several orders of magnitude with high efficiency.

Using the results from this work, users can build their own Peer-to-Peer applications. The system enhances simulation speed and even provides higher availability via more users joining the network. Our work demonstrates that existing Peer-to-Peer technologies can be optimized for distributed simulation or MMVE domains and fulfills all the requirements in Section 1.1.

Bibliography

[1] M. Esch, J. Botev, H. Schloss, and I. Scholtes, "Gp3-a distributed grid-based spatial index infrastructure for massive multiuser virtual environments," presented at Parallel and Distributed Systems, 2008. ICPADS'08. 14th IEEE International Conference on, 2008.

[2] J. Botev, A. Hohfeld, H. Schloss, I. Scholtes, P. Sturm, and M. Esch, "The HyperVerse: concepts for a federated and Torrent-based'3D Web'," *International Journal of Advanced Media and Communication*, vol. 2, pp. 331-350, 2008.

[3] A.-T. Stephanos and S. Diomidis, "A survey of peer-to-peer content distribution technologies," *ACM Comput. Surv.*, vol. 36, pp. 335-371, 2004.

[4] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*. San Diego, California, USA: ACM, 2001.

[5] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Middleware 2001*: Springer, 2001, pp. 329-350.

[6] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*. San Diego, California, USA: ACM, 2001.

[7] S. Ratnasamy, "A Scalable Content-Addressable Network," University of California at Berkeley, 2002.

[8] R. Steinmetz and K. Wehrle, *Peer-to-Peer systems and applications*, 1st ed: Springer, 2005.

[9] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, and W. Weimer, "Oceanstore: An architecture for global-scale persistent storage," *ACM Sigplan Notices*, vol. 35, pp. 190-201, 2000.

[10] A. Oram, "Peer-to-Peer Makes the Internet Interesting Again," 2000.

[11] R. Steinmetz and K. Wehrle, "Peer-to-peer-networking &-computing," *Informatik-Spektrum*, vol. 27, pp. 51-54, 2004.

[12] M. Knoll, *Geostry: A Peer-to-Peer System for Location-based Information*: Suedwestdeutscher Verlag fuer Hochschulschriften.

[13] T. S. E. Ng and Z. Hui, "Towards global network positioning," in *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*. San Francisco, California, USA: ACM, 2001.

[14] S. Saroiu, P. K. Gummadi, and S. D. Gribble, "Measurement study of peer-to-peer file sharing systems," presented at Electronic Imaging 2002, 2001.

[15] OpenNap, "Open Source Napster Server," http://opennap.sourceforge.net/, 2001.
[16] "TheNapsterHomepage", " http://www.napster.com."
[17] R. Matei, A. Iamnitchi, and P. Foster, "Mapping the Gnutella network," *Internet Computing, IEEE*, vol. 6, pp. 50-57, 2002.
[18] Clip2/TheGnutellaDeveloperForum(GDF), "The Annotated Gnutella Protocol Specification v0.4."
[19] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong, "Freenet: A distributed anonymous information storage and retrieval system," presented at Designing Privacy Enhancing Technologies, 2001.
[20] B. Loban, "Between rhizomes and trees: P2P information systems," *First Monday*, vol. 9, 2004.
[21] T. Clingberg and R. Manfredi, "Gnutella0.6," 2002.
[22] G. Li, "JXTA: A network programming environment," *Internet Computing, IEEE*, vol. 5, pp. 88-95, 2001.
[23] J. Liang, R. Kumar, and K. W. Ross, "The FastTrack overlay: A measurement study," *Computer Networks*, vol. 50, pp. 842-858, 2006.
[24] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim, "A survey and comparison of peer-to-peer overlay network schemes," *IEEE Communications Surveys and Tutorials*, vol. 7, pp. 72-93, 2005.
[25] K. Hildrum, J. D. Kubiatowicz, S. Rao, and B. Y. Zhao, "Distributed object location in a dynamic network," *Theory of Computing Systems*, vol. 37, pp. 405-440, 2004.
[26] K. Aberer, "P-Grid: A self-organizing access structure for P2P information systems," presented at Cooperative Information Systems, 2001.
[27] D. Malkhi, M. Naor, and D. Ratajczak, "Viceroy: A scalable and dynamic emulation of the butterfly," presented at Proceedings of the twenty-first annual symposium on Principles of distributed computing, 2002.
[28] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the xor metric," in *Peer-to-Peer Systems*: Springer, 2002, pp. 53-65.
[29] M. Naor and U. Wieder, "Novel architectures for P2P applications: the continuous-discrete approach," *ACM Transactions on Algorithms (TALG)*, vol. 3, pp. 34, 2007.
[30] M. Naor and U. Wieder, "A simple fault tolerant distributed hash table," in *Peer-to-Peer Systems II*: Springer, 2003, pp. 88-97.
[31] M. Naor and U. Wieder, "Scalable and dynamic quorum systems," *Distributed Computing*, vol. 17, pp. 311-322, 2005.
[32] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron, "Topology-aware routing in structured peer-to-peer overlay networks," in *Future directions in distributed computing*: Springer, 2003, pp. 103-107.
[33] A. Rowstron and P. Druschel, "Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility," presented at ACM SIGOPS Operating Systems Review, 2001.

[34] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron, "One ring to rule them all: service discovery and binding in structured peer-to-peer overlay networks," presented at Proceedings of the 10th workshop on ACM SIGOPS European workshop, 2002.

[35] W. Ding and S. S. Iyengar, "Bootstrapping chord over manets-all roads lead to rome," presented at Wireless Communications and Networking Conference, 2007. WCNC 2007. IEEE, 2007.

[36] M. Knoll, A. Wacker, G. Schiele, and T. Weis, "Decentralized bootstrapping in pervasive applications," presented at Pervasive Computing and Communications Workshops, 2007. PerCom Workshops' 07. Fifth Annual IEEE International Conference on, 2007.

[37] M. Knoll, A. Wacker, G. Schiele, and T. Weis, "Bootstrapping in peer-to-peer systems," presented at Parallel and Distributed Systems, 2008. ICPADS'08. 14th IEEE International Conference on, 2008.

[38] C. Cramer, K. Kutzner, and T. Fuhrmann, "Bootstrapping locality-aware P2P networks," presented at Networks, 2004.(ICON 2004). Proceedings. 12th IEEE International Conference on, 2004.

[39] Y. Chawathe, S. Ramabhadran, S. Ratnasamy, A. LaMarca, S. Shenker, and J. Hellerstein, "A case study in building layered DHT applications," presented at ACM SIGCOMM Computer Communication Review, 2005.

[40] M. Knoll and T. Weis, "A P2P-Framework for Context-based Information," in *1st International Workshop on Requirements and Solutions for Pervasive Software Infrastructures (RSPSI) at Pervasive*: Citeseer, 2006.

[41] S. Domnitcheva, "Location modeling: State of the art and challenges," presented at Proceedings of the Workshop on Location Modeling for Ubiquitous Computing, 2001.

[42] H. Sagan, *Space-filling curves*, vol. 18: Springer-Verlag New York, 1994.

[43] H. V. Jagadish, "Linear clustering of objects with multiple attributes," in *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*. Atlantic City, New Jersey, USA: ACM, 1990.

[44] J.-M. Wierum, "Logarithmic path-length in space-filling curves," presented at CCCG, 2002.

[45] G. D. Abowd, A. K. Dey, P. J. Brown, N. Davies, M. Smith, and P. Steggles, "Towards a better understanding of context and context-awareness," presented at Handheld and ubiquitous computing, 1999.

[46] A. Dix, T. Rodden, N. Davies, J. Trevor, A. Friday, and K. Palfreyman, "Exploiting space and location as a design framework for interactive mobile systems," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 7, pp. 285-321, 2000.

[47] C. A. Furuti, "Map projections," http://www.progonos.com/furuti/MapProj/.

[48] C. Gotsman and M. Lindenbaum, "On the metric properties of discrete space-filling curves," *Image Processing, IEEE Transactions on*, vol. 5, pp. 794-797, 1996.

[49] M. Knoll and T. Weis, "Optimizing locality for self-organizing context-based systems," in *Self-Organizing Systems*: Springer, 2006, pp. 62-73.

[50] H. Sagan, "Hilbert's Space-Filling Curve," in *Space-Filling Curves*: Springer, 1994, pp. 9-30.

[51] R. Niedermeier, K. Reinhardt, and P. Sanders, "Towards optimal locality in mesh-indexings," presented at Fundamentals of Computation Theory, 1997.

[52] C. Schmidt and M. Parashar, "Flexible information discovery in decentralized distributed systems," presented at High Performance Distributed Computing, 2003. Proceedings. 12th IEEE International Symposium on, 2003.

[53] S. Zhou, G. R. Ganger, and P. A. Steenkiste, "Location-based node ids: Enabling explicit locality in dhts," 2003.

[54] H. Ballani and P. Francis, "Towards a deployable ip anycast service," presented at Proceedings of the Workshop on Real, Large Distributed Systems, 2004.

[55] S. Holzapfel, S. Schuster, and T. Weis, "VoroStore--A Secure and Reliable Data Storage for Peer-to-Peer-Based MMVEs," presented at 2011 IEEE 11th International Conference on Computer and Information Technology (CIT), 2011.

[56] D. Heutelbeck, "Distributed space partitioning trees and their application in mobile computing," Fernuniv., Fachbereich Informatik, 2005.

[57] D. Heutelbeck and M. Hemmje, "RectNet-A Distributed Geometrical Data Structure," presented at Mobile Data Management, 2006. MDM 2006. 7th International Conference on, 2006.

[58] J. Kubiatowicz, "The OceanStore Project," 2011.

[59] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer, "Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs," presented at ACM SIGMETRICS Performance Evaluation Review, 2000.

[60] M. Waldman, A. D. Rubin, and L. F. Cranor, "Publius: A Robust, Tamper-Evident Censorship-Resistant Web Publishing System," presented at 9th USENIX Security Symposium, 2000.

[61] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker, "Application-level multicast using content-addressable networks," in *Networked Group Communication*: Springer, 2001, pp. 14-29.

[62] Z. Li and T. Weis, "Using zone code to manage a Content-Addressable Network for Distributed Simulations," in *2012 IEEE 14th International Conference on Communication Technology (ICCT)*: IEEE, 2012, pp. 1350-1357.

[63] E. W. Zegura, K. L. Calvert, and S. Bhattacharjee, "How to model an internetwork," presented at INFOCOM'96. Fifteenth Annual Joint Conference of the IEEE Computer Societies. Networking the Next Generation. Proceedings IEEE, 1996.

[64] Z. Xu and Z. Zhang, "Building low-maintenance expressways for p2p systems," *Hewlett-Packard Labs, Palo Alto, CA, Tech. Rep. HPL-2002-41*, 2002.

[65] O. D. Sahin, D. Agrawal, and A. El Abbadi, "Techniques for efficient routing and load balancing in content-addressable networks," presented at Peer-to-Peer Computing, 2005. P2P 2005. Fifth IEEE International Conference on, 2005.

[66] D. Boukhelef and H. Kitagawa, "Multi-ring infrastructure for content addressable Networks," in *On the Move to Meaningful Internet Systems: OTM 2008*: Springer, 2008, pp. 193-211.

[67] D. Boukhelef and H. Kitagawa, "Dynamic load balancing in RCAN content addressable network," in *Proceedings of the 3rd International Conference on Ubiquitous Information Management and Communication*. Suwon, Korea: ACM, 2009, pp. 98-106.

[68] Z. Li and T. Weis, "Content-Addressable Network for Distributed Simulations.," presented at 2013 IEEE International Conference on Computer Science and Automation Engineering (CSAE 2013), Guangzhou China, 2013.

[69] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, pp. 1098-1101, 1952.

[70] X. Sun, "SCAN: a small-world structured P2P overlay for multi-dimensional queries," presented at Proceedings of the 16th international conference on World Wide Web, 2007.

[71] Z. Li and T. Weis, "CAN Tree Routing for Content-Addressable Network," *Sensors & Transducers*, vol. 162, pp. 124-130, 2014.

[72] D. Heutelbeck, R. Raeth, and C. Unger, "Fault tolerant geographical addressing," in *Innovative Internet Community Systems*: Springer, 2003, pp. 144-155.

Curriculum Vitae

Der Lebenslauf ist in der Online-Version aus Gründen des Datenschutzes nicht enthalten.

Index