uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Stevan Zonjic
AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M.C.S.
GRADE / DEGREE

School of Information Technology and Engineering
FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Multidimensional Query Bases Routing for Massively Multiuser Virtual Environments
TITRE DE LA THÈSE / TITLE OF THESIS

S. Shirmohammadi
DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

A. Nayak

C. Horn Lung

H. Mouftah

Gary W. Slater
Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

Multidimensional Query Based Routing for Massively Multiuser Virtual Environments

by Stevan Zonjic

A thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
In partial fulfillment of the requirements for the degree of

Master of Computer Science

Ottawa-Carleton Institute for Computer Science
School of Information Technology and Engineering

Faculty of Engineering
University of Ottawa

Your file Votre référence
ISBN: 978-0-494-61348-1
Our file Notre référence
ISBN: 978-0-494-61348-1

Abstract

In recent years, massively multi-user virtual environments (MMVE) have been a major research topic for the virtual reality community. In order for a MMVE to be highly responsive and to progress seamlessly, entities interacting in the shared virtual environment must update each other frequently. Every time there is a change related to an entity's state, an update message needs to be sent to inform other entities in the shared virtual environment. This thesis proposes a new architecture based on Z-order curve and KD-tree that can efficiently handle update message distribution to interested entities' locations in three-dimensional (3D) virtual environments where locations are naturally described in terms of 3D coordinates. We also aim to minimize the number of routing hops in distribution of update messages to minimize end-to-end delay, as required in MMVEs, especially when updates are forwarded to a specific range of users/entities in the MMVE.

Acknowledgements

I would like to express sincere gratitude to my thesis supervisor Dr. Shervin Shirmohammadi, for providing valuable support and guidance. Also, I would like to thank Behnoosh Hariri for providing me with ideas, advice, and assistance during my research of the subject presented in this thesis.

Finally, I would like to thank to my family for giving me love and support.

Table of Contents

List of Figures

List of Tables

List of Acronyms

Acronym	Definition
BSP	Binary Space Partitioning
CAN	Content Addressable Network
DHT	Distributed Hash Table
MMOG	Massively Multiuser Online Game
MMVE	Massively Multiuser Virtual Environment
MSB	Most Significant Bit
P2P	Peer-to-Peer
VE	Virtual Environment
VERA	Virtual Environment Routing Algorithm

Chapter 1

Introduction

1.1 Motivation

In recent years, massively multi-user virtual environments (MMVE) have been a major research topic for the virtual reality community ([4], [6], [7], [8], [9]). MMVE applications incorporate computer graphics and sound simulation, together with the achievements in the networking field that further motivated multi-user interaction over the internet, to simulate the experience of real-time interaction between multiple users in a shared three-dimensional (3D) virtual world.

In MMVE, each user runs an interactive interface program on a "client" computer connected to a wide-area network which simulates the experience of immersion in a virtual environment by rendering images and sounds of the environment as perceived from the user's simulated viewpoint. Also, each user is represented by an entity rendered on every other user's computer, and multi-user interaction is supported by matching user actions to entity updates (i.e. motion/sound generation) in the shared virtual environment.

Multi-user virtual environment technology can be applied in many areas, for example, distributed training, simulation, education, home shopping, virtual meetings, and massively multi player online games (MMOGs). More specifically, let us think about a virtual city where multiple users interact in real-time environment. As each user moves through the city, a graphical representation of that user is displayed moving on each other user's screen. Also, if any user talks into a microphone, the voice is played so as to appear to come from the position of the entity representing the user. This concept is more appealing for social interactions and commerce comparing to textual or two-dimensional multimedia interfaces used by chat programs and browsers for the World Wide Web.

One example of the importance of MMVEs is the success of MMOGs. Due to increasing broadband adoption among consumers and relatively cheap high-bandwidth internet connections that allow large number of players to play together, together with the advancement of computer graphics and artificial intelligence, MMOGs have grown dramatically and become a profitable sector to vendors. In the last couple of years, studies have revealed that online games are becoming a major contributor to Internet traffic. DFC Intelligence forecasts that the world online game market will expand to over $13 billion by 2011 and the number of online gamers in the 20 leading online gaming countries is expected to increase by 51%.

In order for the MMVE to be highly responsive and to progress seamlessly, entities interacting in the shared virtual environment must update each other frequently. Every time there is a change related to entity's state or a modification to the shared virtual environment or impact on the other entities, an update message needs to be sent to inform other entities in the shared virtual environment. With a significant increase of virtual environment users and entities, there is a related increase in the number of update messages that are being exchanged among the users and entities. As a result, there is a major increase in the importance of effective update messages distribution on the overlay framework of the virtual environment, while achieving an acceptable level of latency is also a concern in the update messages exchange process [1].

1.2 Research Problem

MMVEs are interactive computer simulations that immerse users in an alternate, yet believable reality, where multiple programs on different machines have to be managed, and participants' communication must be coordinated. Currently, there are two MMVE communication models for implementation of distributed MMVEs, specifically, centralized model and distributed model.

In centralized model, one computer (server) collects all of the data from different clients, stores changes in some collection of data structures (the centralized database), and then

coordinates communication by sending the results back out to each participating client (Figure 1). Each client then renders the scene and handles user input.

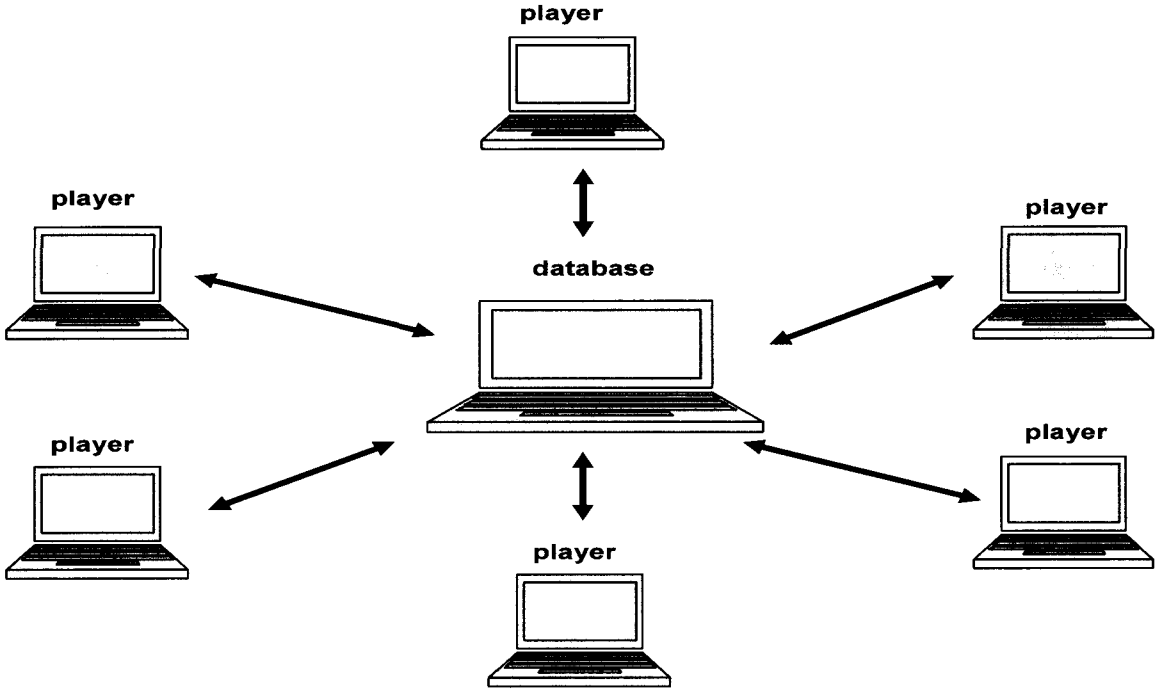


Figure 1.Centralized model

The primary advantage of this model is that message distribution responsibility is transferred from clients to servers, so clients do not have to perform a lot of processing, storage or messaging to maintain consistent state among entities in a large virtual environment. However, this approach is not very scalable, since, as more and more clients enter the virtual environment, they must wait increasingly longer due to the bottleneck on the centralized server. [2]

On the other hand, an alternative, more scalable approach is to incorporate a distributed model, i.e. peer-to-peer (P2P) topology (Figure 2). In this model, each client maintains its own complete, local copy of the database as well as performing rendering, computation and animation of objects. Clients communicate over a network where each client can send

messages directly to any other client, so when there is a change in client's database, it sends the update data out so that other clients can update their individual databases.

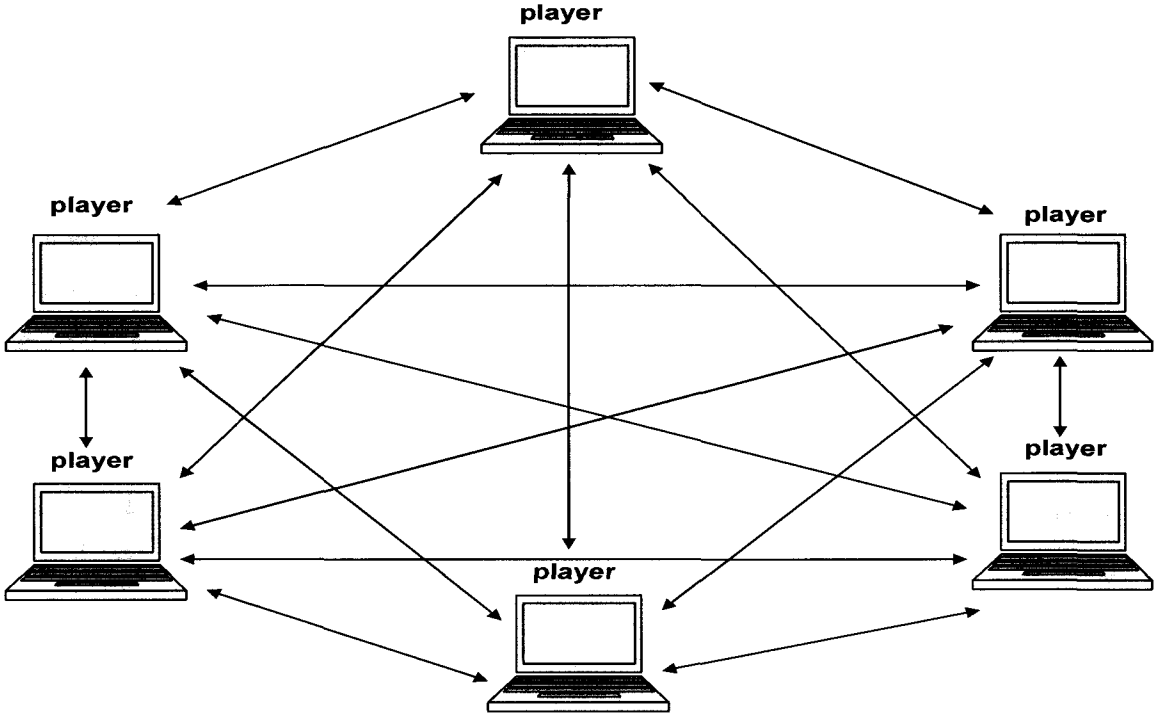


Figure 2.Distributed model

This changes the scalability problem from a CPU bottleneck (in the centralized model) to one where there are many connections and messages.

In a P2P system design, a set of peers communicates over a network where each peer can send messages directly to any other peer. If this network is a unicast network (i.e. it only supports unicast messages), peers send a unicast message to other peers every time an entity is updated. This concept is shown in Figure 3. In order to have a scalable system that supports many simultaneous users, peers can maintain lists of the entities resident in each cell and send update messages only to other peers managing an entity residing in a cell visible to the cell in which an update occurred. Therefore, with this approach each peer receives only a subset of all update messages. On the other hand, this design does not scale infinitely, since all peers must maintain an up-to-date mapping of which entities are in each

cell. Therefore, in order to have synchronized mapping among peers, update messages are sent to all peers whenever an entity moves into a new cell, where number of these update messages may be relatively small, but it grows to O(NP) for N entities and P peers.[2]

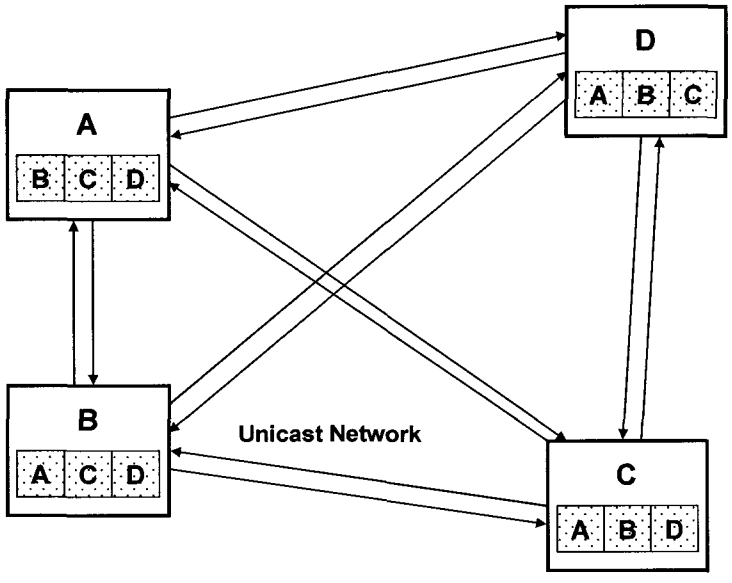


Figure 3.Peer-to-peer topology with a unicast network

In a P2P system design, if a network supports messages multicasting, then peers can send a single multicast message to a subset of peers all at once (shown in Figure 4).

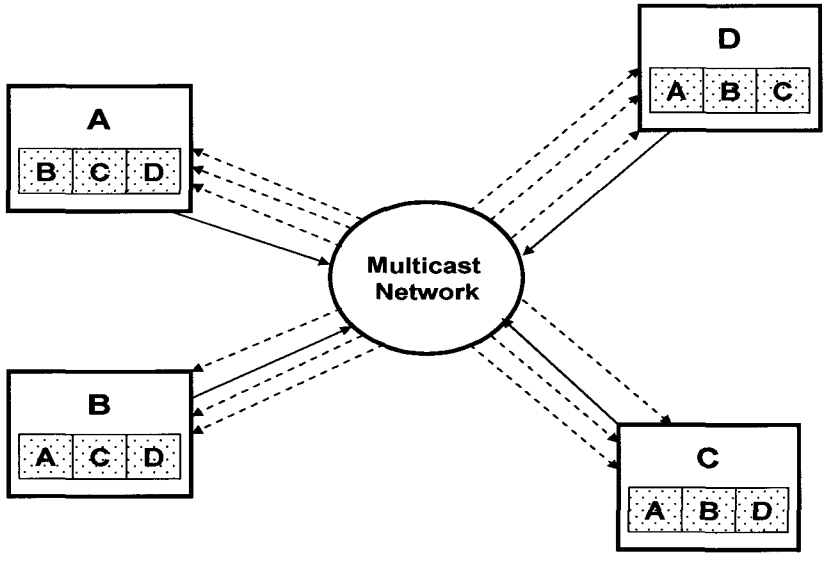


Figure 4.Peer-to-peer topology with a multicast network

A multicast group can be assigned to each cell, and when update occurs, a peer sends a message to the multicast group representing the cell in which the update occurred, while all peers listen to the multicast groups representing the cells visible to the cells containing their entities. With this architecture, peers do not need to maintain explicit lists of entities residing in each cell. Instead, they join and leave multicast groups as their entities move between cells, causing implicit messages to be generated by the multicast network to update routing tables. This design also does not scale infinitely, since the number of multicast membership changes grows with O(N). [2]

MMOGs are natural applications for peer-to-peer overlays, since scalability is an important factor when a lot of users play simultaneously in huge virtual worlds. P2P overlay networks provide scalability by moving the computational load and storage requirements from central server clusters to peers in the network. With this architecture in place, each user's computer while being connected to the game also automatically provides additional resources to the game which reduces requirements for a powerful central server clusters, and at the same time reduces the cost of setting up and maintaining massive multiplayer online games [4].

However, there is one disadvantage of using peer-to-peer networks for MMOGs, specifically, the lack of a central authority that regulates access and prevents cheating. Also, depending on the type of communication used for player updates, P2P architecture can suffer from bottlenecks and lack of scalability. For example, an early P2P game MiMaze [5] uses all-to-all communications through IP multicasting for player updates which does not scale well and suffers from bottlenecks. Therefore, the communication must only occur within subsets of distributed entities, specifically, mutable spatial objects in the virtual world, while allowing for a global sense of existence in a larger setting. In order to have good performance and scalability of the game, it is necessary to successfully determine these subsets, often referred to as interest management, from a globe of many participants. These interest management areas can be roughly classified as either neighbour based or region based. In neighbour based interest management, peers maintain a list of nearby entities with

which to communicate, whereas in region based interest management updates are performed through queries on regions.

Update messages distribution among entities in the MMVE plays a major role in overlay network scalability and flexibility. In MMVEs, update messages are usually destined to objects in its proximity, so MMVE networks require a sophisticated addressing mechanism which minimizes network overhead and improves network latency and scalability.

The main focus of our research is to develop a routing algorithm that will improve the update messages exchange process by reducing network latency, since low update delay is critically important for MMVEs (like MMOGs), and improving network scalability, since MMVEs often host thousands of participants at any time.

1.3 Research Objective

In recent years massive multiplayer online games (MMOGs) have been highly popular. Most existing MMOGs are role playing games or real-time strategy games. Typical examples of MMOGs include EverQuest, Ultima Online, There.com, Star Wars Galaxies, The Sims Online, Simcountry, etc. The basic characteristic of most MMOGs is that the player assumes the role of a character in a virtual world. More specifically, MMOGs are modeled as a set of interacting entities in a 3D space, where a large population of players interact with entities and each other, i.e. using avatar entities [3].

Each year there is a considerable growth in number of users as well as in the size of the virtual worlds. With this growth comes a significant increase of the requirements for server hardware.

Nowadays MMOGs provider typically has to deal with a problem of serving thousands of users with entire server clusters. Peer-to-peer networks have the attributes to effectively support MMOGs. With their high scalability and flexibility, P2P meet the requirements of

connecting hundreds of thousands of users all over the world without a central server, while the network bandwidth requirements remain at a reasonable level.

This research is based on the fact that update messages are transferred to entities whose visibility range lies within an object's area of interest. Recently, different P2P architectures have been proposed to tackle the scalability issue of the MMOGs ([3], [4], [5], [6], [7]) which is out of the scope of this thesis. A novel approach to support exchange of update messages among massive number of users could be to index MMVE entities according to objects' positions and think of them as composing a distributed database, where virtual environment objects are considered as records indexed with a number of keys. In this specific case MMVE entities 3D coordinates are used as keys for indexing. This is explained next.

In distributed databases users tend to submit some form of a range query where the user specifies a range restriction on one or more dimensions. For example, with a three attributes key set, the retrieval algorithms could be required to retrieve all those records for two specific values of the key set but with any value for the other one; or records whose key set attributes lie within specific range. This sort of a request is similar to interest management communication, where communication is only required with a specific set of clients in MMVE. Multidimensional indexing is based on the notion that more than one attribute of an object should be constituting primary key set for that object and that all attributes should have an equal importance in determining the placement of a record in a storage, and consequently in optimization issues. We believe that this notion can be applied to MMVE routing and index searching, and it can increase efficiency of update message exchange. To summarize, this research has the following objectives:

- Efficient update messages distribution between entities in MMVEs

  Considering that entities' area of interest usually lies within certain visibility range, the routing algorithm for messages distribution should be based on physical entities' positions

- Increasing the efficiency of the routing algorithm for messages distribution by incorporating entity's physical position, since that entity's area of interest usually lies within certain visibility range.

Considering these requirements, we proposes an architecture that uses Z-order space filling curve to perform multi-dimensional indexing and a routing algorithm that is based on traversing the Z-order curve in the KD-tree format.

To achieve the above objectives, this thesis makes use of certain techniques and approaches. For example, space filling curves that are used to map a multidimensional space on to a one-dimensional space. Space filling curves pass through every point in space once and have a one-to-one correspondence between the coordinates of the points and the one-dimensional sequence numbers of the points on the curve. Z-order curve is one of the space filling curves that is used in mapping process from multiple dimensions to one dimension. The motivation behind usage of Z-order curve for indexing is that it has a good clustering among multidimensional points along a one dimensional mapping, a feature desirable when searching for compact ranges.

Also, we choose a KD-tree, a multidimensional binary search tree, to represent a Z-order curve, since KD-trees have been shown to be well suited for range searching and they also group the data points together alternating the discriminating keys. With this architecture, we can minimize the number of hops in distribution of update messages, especially when updates are forwarded to a specific range of users/entities in the virtual environment.

1.4 Research Contributions

This thesis contains a number of contributions, including:

- Novel approach for applying multidimensional indexing and multidimensional binary search tree to ID assignment in MMVEs.

- Design and implementation of a novel routing algorithm, called VERA (Virtual Environment Routing Algorithm), for MMVEs based on Z-order space filling curve and KD-tree, using peers' physical network proximity.

- An approach for handling network dynamics, specifically, join/movement/departure process for this newly proposed algorithm

- Proof of concept, implementation with performance evaluation and validation of the design and theory

In addition, the following peer-reviewed publications have resulted from this work:

- (submitted) S. Zonjic, B. Hariri, S. Shirmohammadi, "Location Based P2P Overlays for MMVEs: A Database Perspective", IEEE Trans. On Parallel and Distributed Systems.

- (accepted) S. Zonjic, B. Hariri, S. Shirmohammadi, "Multidimensional Query Based Routing for Virtual Environments," IEEE International Conference on Virtual Environments, Human-Computer Interfaces, and Measurement Systems, (VECIMS 2009), Hong Kong, China, May 2009.

1.5 Organization of the Thesis

The rest of this thesis is organized as outlined below:

Chapter 2 Background and Related Work discusses related P2P architectures and their routing algorithms. In particular, we look in details at some of these architectures, and examine their strengths and weaknesses for our application domain. Specifically, we present following P2P architectures: first generation P2P architectures, i.e. Napster and Gnutella, and several new generation of scalable peer-to-peer systems that support a distributed hash

table (DHT) functionality (Tapestry, Pastry, Chord, Content Addressable Networks (CAN), Kademlia, and Viceroy).

Chapter 3 Applying Multidimensional Indexing and Multidimensional Binary Search Tree to MMVEs presents multidimensional indexing, i.e. the method of mapping the points of the multidimensional space to a one-dimensional index, and gives a detailed description of multidimensional binary search tree, i.e. the KD-tree. Multidimensional indexing is based on the fact that one record can have multiple attributes specified as constituting the primary key set for that record, and that all keys should have the same importance in determining the physical placement of a record in store and consequently in optimization issues. We give a detailed description of one of the space-filling curves that is used for multidimensional indexing, specifically the Z-order curve. We also show KD-tree suitability for representation of multidimensional indexing using Z-order space filling curve, and present different querying options with KD-tree data structure.

Chapter 4 VERA: Virtual Environment Routing Algorithm presents a P2P routing algorithm for MMVEs based on Z-order space filling curve and KD-tree. Specifically, it shows how Z-order curve is used to map 3-dimensional points to one-dimensional indexes, and how these points are stored in proper order (as mapped by Z-order space filling curve) in the KD-tree data structure. Also it shows how routing tables are constructed based on the proposed architecture, and how the information available in each node's routing table is used to perform efficient routing between points in 3-dimensional virtual environment, i.e. the routing algorithm.

Chapter 5 Supporting Network Dynamics describes how user movements are supported by the proposed architecture. More specifically, it explains the routing table update process in three important situations, i.e. when nodes join the network, move in the network, and leave the network.

Chapter 6 Performance Evaluation presents analysis of preliminary experimental results obtained from the initial architecture development.

Chapter 7 Conclusions and Future Work summarizes and concludes the thesis and provides recommendations for future research.

Chapter 2

Background and Related Work

2.1 Introduction

Even thought P2P systems were introduced only few years ago, many research projects have been conducted in the area of scalable and decentralized distributed applications ([12], [13], [14], [15], [16], [17]). Structured P2P overlay networks offer a novel platform for a variety of decentralized distributed applications. They provide efficient fault tolerant routing, object location and load balancing within a self-organizing overlay network, so many projects aim at constructing peer-to peer overlays and understanding more of the issues and requirements of such applications and systems. One of the major problems in large scale peer-to-peer applications is to provide efficient algorithms for object location and routing strategy for message exchange.

In this section some of the existing peer-to-peer architectures and routing algorithms are presented. We will look in details at some of these architectures, and examine their strengths and weaknesses for our application domain.

First generation P2P systems Napster and Gnutella are presented first. These P2P systems have significant scaling problem, i.e. Napster has a centralized directory service, and Gnutella employs a flooding based querying mechanism that is not suitable for large systems ([10], [11]).

On the other hand, several research groups addressed the significant scaling problem in the initial designs for peer-to-peer systems (Napster, Gnutella, etc.) by proposing a new generation of scalable peer-to-peer systems that support a distributed hash table (DHT) functionality (Tapestry ([18], [19]), Pastry [20], Chord [21], Content Addressable Networks

(CAN) [24], Kademlia [25], Viceroy [26], etc.), where files are associated with a key (produced, for example, by hashing the file name) and each node in the system is responsible for storing a certain range of keys.

Although the above approaches improve the initial limitations of traditional P2P systems, they have their shortcomings. None of the above approaches is able to do range searches based on peers' actual network proximity. Also, none of them uses objects' physical location to do efficient routing. Let us now look into details of each of these P2P architectures.

2.2 Napster

Napster is one of the first generation P2P networks, and is better known as a music exchange system. It has a constantly updated object directory maintained at central Napster server, which stores the index of all the files available within the Napster user community. In order to retrieve a file a node queries this central server to find which other nodes hold their desired file, obtains the IP address of the node storing requested file, and finally downloads the desired file directly from this node. Figure 5 presents the architecture of centralized P2P network like Napster ([10], [30]).

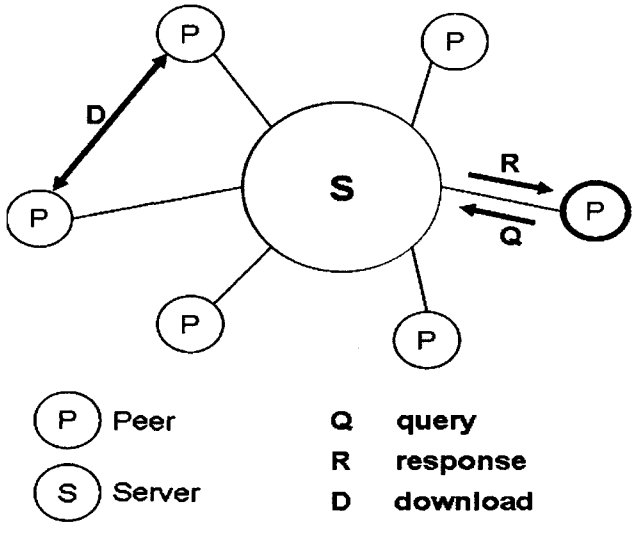


Figure 5.Napster P2P centralized network

Although Napster uses a peer-to-peer communication model for the actual file transfer, and avoids query routing and other problems of other P2P systems, the process of locating a file is still very much centralized. This approach has both very poor scalability (i.e. it is very expensive), and high vulnerability (since there is a single point of failure).

2.3 Gnutella

Gnutella is a decentralized and unstructured P2P network with neither a centralized directory nor any precise control over the network topology or object placement. Nodes in Gnutella network self-organize into a virtual mesh network on which requests for a file are flooded with a certain scope. In order to join Gnutella network, a node must connect to a known Gnutella node to get a lists of some existing Gnutella nodes. To find a file, a node typically queries by flooding (presented in Figure 6), i.e. issuing a query to all its neighbours within a certain scope ([11], [30]).

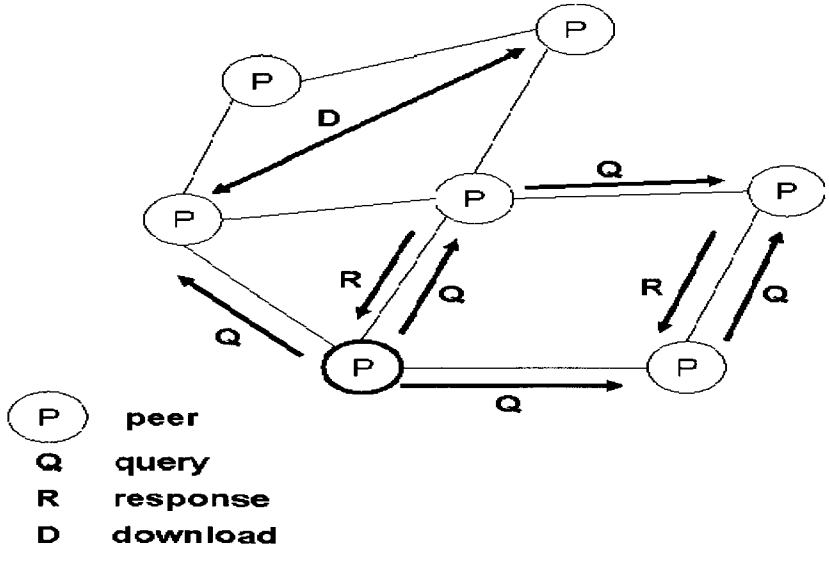


Figure 6.Querying by flooding (Gnutella network)

Obviously, the flooding based lookup mechanism in not scalable and, because the flooding has to be reduced at some point, may fail to locate the information that is actually in the system.

2.4 Plaxton Algorithm

Plaxton algorithm is one of the first routing algorithms that could be used by DHTs. In Plaxton each node can interchange between the roles of servers (where objects are stored), routers (which forward messages) and clients (origins of requests). This data structure is also called a Plaxton mesh. In a Plaxton mesh every destination node is the root node of its own tree, which is a unique spanning tree across all nodes. Any leaf can traverse a number of intermediate nodes en route to the root node; therefore, the Plaxton mesh of neighbour maps is a large set of embedded trees in the network, one rooted at every node (Figure 7. presents an example of Plaxton routing).

Although it was not intended for use in P2P systems (assumes a relatively static data structure, without node or object insertions or deletions), this algorithm provides a very efficient routing. It works by "correcting" one digit at a time: if node number 0325 received a lookup query with a key 4598, it uses local routing maps at each node (neighbour maps) to incrementally route overlay messages to the destination ID digit by digit (e.g. ***8 → **98 → *598 → 4598, where *'s represent any digit) [12].

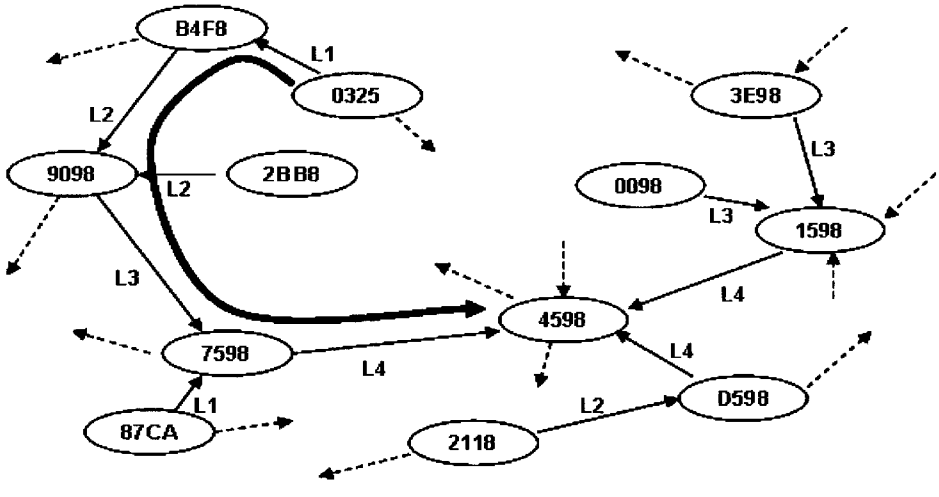


Figure 7.Plaxton routing example

Figure 7 shows a path taken by a message originating from node 0325 destined for node 4598 in a Plaxton mesh using hexadecimal digits of length 4 (65536 nodes in namespace). In order to achieve this, a node must have, as neighbours, nodes that match each prefix of its own identifier but differ in the next digit.

Therefore, for a system of n nodes, each node has $O(\log n)$ neighbours, and since one digit is corrected each time query is forwarded, it routes in at most $O(\log n)$ hops.

The Plaxton location and routing system offers several advantageous characteristics for both routing and location [12].

- Simple Fault Handling – in case of any single node or server failure, it is possible to route around by choosing another node with a similar suffix (since routing only requires nodes that match a certain suffix).

- Scalability – the system is decentralized, i.e. all routing is done using locally available data.

- Exploiting Locality – converges rapidly, since the number of nodes to route drops geometrically with each additional hop (each additional digit of a suffix reduces the number of satisfying candidates by a factor of the ID base b).

- Proportional Route Distance – the total network distance traveled by a message during both location and routing phases is proportional to the underlying network distance, so the routing on the Plaxton overlay incurs a reasonable overhead.

Nevertheless, there are serious limitations to the original Plaxton method:

- Global Knowledge – in order to achieve unique mapping, the Plaxton method requires global knowledge at the time that the Plaxton mesh is constructed, which complicates the process of adding and removing nodes from the network.

- Root Node Vulnerability – root node is a single point of failure, since it is the node that every client relies on to provide an object's location information.

- Inadaptable – it lacks the ability to adapt to dynamic query patterns, such as distant hotspots. Insertions could be only handled by using global knowledge to recompute the function for mapping objects to root nodes.

2.5 Tapestry

Tapestry, another self-organizing routing and object location system, is a modification of the Plaxton algorithm designed to adapt a dynamic node population. It provides high performance, scalability, and location-independent routing of messages to close-by endpoints, using only localized resources.

Each node in Tapestry is assigned uniform nodeId randomly from a large identifier space of 160-bit values with a globally defined radix (e.g. hexadecimal, yielding 40-digit identifiers) [18]. Tapestry assumes that nodeIds are approximately evenly distributed in the namespace, which can be accomplished by using any secure hashing algorithm [23].

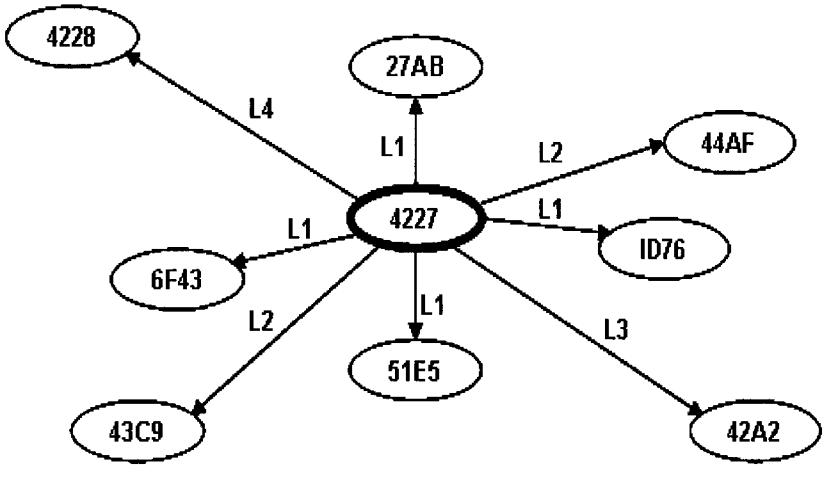


Figure 8.Tapestry routing mesh from the perspective of a single node

In order to deliver messages, each node in Tapestry maintains a routing table consisting of nodeIds and IP addresses of the neighbour nodes (the nodes with which the local node communicates). The routing is performed by forwarding messages across neighbour links to nodes whose nodeIds are progressively closer (i.e. matching larger prefixes) to the destination ID in the ID space. Tapestry routing mesh contains each node's local tables, called neighbour maps, to route overlay messages to the destination ID digit by digit. Each node has a neighbour map with multiple levels, where each level contains links to nodes matching a prefix up to a digit position in the ID, and contains a number of entries equel to the ID's base. Figure 8 shows some of the outgoing neighbour links of a node, where higher level entries match more digits. Together, these links form the local routing table for the particular node. [18] [19]

If node number 5230 received a lookup query with a key 42AD, it uses local routing maps at each node (neighbour maps) to incrementally route overlay messages to the destination ID digit by digit (e.g. 4*** → 42** → 42A* → 42AD, where *'s represent any digit). Figure 9 shows a path taken by a message originating from node 5230 destined for node 42AD in a Tapestry routing mesh using hexadecimal digits of length 4. In the case that a digit cannot be matched, Tapestry routes to some live node with the closest ID (process called surrogate routing). The algorithm preserves original properties of having $O(\log n)$ neighbours and routing within $O(\log n)$ hops. [19]

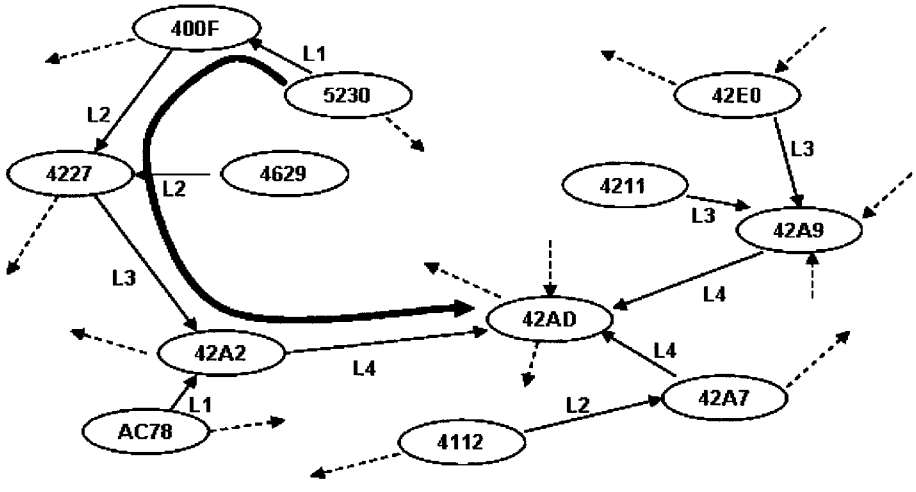


Figure 9.Path of a message in a Tapestry mesh

Unlike Plaxton method, Tapestry supports dynamic network environment by continuing to route reliably even when intermediate links are changing or faulty, which is done by exploiting network path diversity in the form of redundant routing paths (more details can be found in [18], [19]). Also, Tapestry includes several mechanisms to maintain routing table consistency and ensure object availability. More specifically [19],

1. Node Insertion

2. Voluntary Node Deletion

3. Involuntary Node Deletion

In order to maintain availability and redundancy, each node uses periodic beacons to detect outgoing link and node failures, which triggers repair of the routing mesh and launches redistribution and replication of object location references. Despite continuous node turnover, Tapestry retains high success rate at routing messages to nodes and objects.

2.6 Pastry

Pastry is self-organizing overlay network of nodes that performs application level routing and object location in a potentially very large overlay network of nodes connected via the Internet. Each node in the Pastry is assigned a 128-bit node identifier (nodeId) used to indicate a node's position in a circular space, which ranges from 0 to 2^{128} -1. The nodeId is assigned randomly when a node joins the system and it is assumed to be generated such that the resulting set of nodeIds is uniformly distributed in the 128-bit nodeId space. Since nodeIds are generated randomly by computing a cryptographic hash of the node's public key or its IP address, there is a high probability that two nodes with adjacent nodeIds are diverse in geography, jurisdiction, network attachment, etc. [20][31]

For a network of N nodes, Pastry can route to the numerically closest node to a given key in less than $\log_2{}^b$ N steps under normal operation (where b is a configuration parameter with

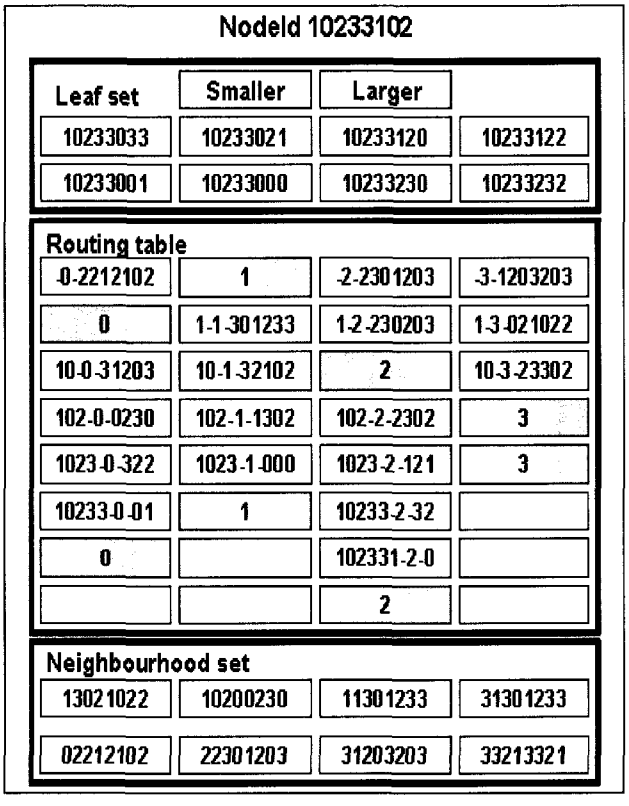
typical value of b = 4). The nodeIds and keys are thought of as a sequence of digits with base 2^b. In Pastry each node is responsible for keys that are numerically closest and it routes messages to the node whose nodeId is numerically closest to the given key. "At each routing step, a node normally sends the message to a node whose nodeId shares with the key a prefix that is at least one digit (or b bits) longer than the prefix that the key shares with the current node's id." [20]

In Pastry every node maintains a routing table and two neighbours sets, one called the Leaf Set (L), which is the set of closest nodes (half with larger Ids, half with smaller Ids), and another called a neighborhood set (M), containing neighbours spread out in the key space. The second set of neighbours is used to achieve more efficient routing when the destination key is outside the Leaf Set. A node's routing table consists of $\log_{2^b} N$ rows with $2^b - 1$ entries each. The $2^b - 1$ entries at row n of the routing table each contain the IP address (together with a nodeId) of a node whose nodeId shares the current node's nodeId in the first n digits, but whose n + 1th digit has one of the $2^b - 1$ possible values other than the n + 1th digit in the current node's nodeId. Since there are potentially many nodes whose nodeId have the appropriate prefix and to have good locality properties; in practice a node is chosen that is close to the present node, according to the proximity metric. In case that there is no node with an appropriate nodeId, the routing table entry is left empty. Since nodes are uniformly distributed to ensure an even population in the nodeId space, on average, only $\log_{2^b} N$ rows are populated in the routing table. [20]

The neighborhood set M includes the nodeIds and IP addresses of the |M| nodes that are closest (according the proximity metric) to the current node, and it is not usually used in messages routing, but for maintaining locality properties. The leaf set L contains nodes that are closest to the current node, where |L| / 2 are numerically closest larger nodeIds and other |L| / 2 are numerically closest smaller nodeIds, relative to the current node's nodeId. Usual values for |M| and |L| are 2^b or 2×2^b. The leaf set is used for message routing. [20]

An example of a hypothetical Pastry node with nodeId 10233102, with b = 2 and L = 8 is presented in Figure 10. All numbers are in base 4, the top row of the routing table is row

zero, the shaded cell in each row of the routing table shows the corresponding digit of the present node's nodeId, and nodeIds in each entry have been split to show the common prefix with 10233102 - next digit - rest of nodeId. The associated IP addresses are not shown. [20]



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	

Neighbourhood set

13021022	10200230	11301233	31301233
02212102	22301203	31203203	33213321

Figure 10.An example of Pastry node

The Pastry routing method is shown in pseudocode form below, and it is executed whenever a message with key K arrives at a node with nodeId A. Let us define some notation. [20]

$\mathbf{R}_l^i$: the entry in the routing table R at column i, $0 \leq i \leq 2^b$ and row l, $0 \leq l < \lfloor 128/b \rfloor$

$\mathbf{L}_i$: the i-th closest nodeId in the leaf set L, $-\lfloor |L|/2 \rfloor \leq I \leq \lfloor |L|/2 \rfloor$, where negative/positive indices specify smaller/larger than the current nodeId, respectively.

$\mathbf{K}_l$: the value of the l's digit in the key K.

shl(A, B): the length of the prefix shared among nodes A and B, in digits.

(1) if ($L_{-\lfloor |L|/2 \rfloor} \leq K \leq L_{\lfloor |L|/2 \rfloor}$) { // K is within range of our leaf set

(2) forward to L_i, such that $| K - L_i |$ is minimal;

(3) } else { // use the routing table

(4) Let l = shl (K, A);

(5) if ($R_l^D \neq$ null) {

(6) forward to R_l^D;

(7) }else { // rare case

(8) forward to T $\in$ L $\cup$ R $\cup$ M, such that

(9) shl(T, K) $\geq l$,

(10) $| T - K | < | A - K |$

(11) }

(12) }

Pastry routing algorithm first checks to see if the key falls within the range of nodeIds covered by its leaf set. If the key falls within that range, the message is forwarded directly to the node in the leaf set whose nodeId is closets to the key, if not, then the routing table is used and the message is forwarded to a node that shares a common prefix with the key by at least one more digit than the current node. In the case that the suitable entry in the routing table is empty or if the appropriate node is not reachable, the message is forwarded to a node that shares a prefix with the key at least as long as the current node, and has a nodeId numerically closer to the key than the current node's id. [20]

This routing method always converges, since each routing step forwards the message to a node that either shares a longer prefix with the key than the current node, or shares a prefix as long as a current node, but has a nodeId numerically closer to the key than the current node. Pastry routing algorithm routes within $O(\log n)$ hops. [20]

When a new node, with nodeId X, wants to join the Pastry network, it first locates a nearby Pastry node A that can be located automatically, for example, by using "expanded ring multicast" or obtaining it by the system administrator through outside channels. After it

locates node A, X asks node A to route a special "join" message with the key equal to X, which will be routed to the existing Pastry node Z whose id is numerically closest to X. As a result of response to receiving a special "join" request, node X receives state tables of nodes A, Z, and all the other nodes that were encountered on the path from A to Z. The new node X examines the information received from these nodes, and then initializes its own state tables (detailed initialization procedure is described in [20]). At the end of the join process, node X informs any nodes that need to be aware of its arrival, so that the state in all other affected nodes is updated. The whole "join" process requires $O(\log_{2b} N)$ exchanged messages. [20]

Node departure or failure in the Pastry network can happen at any time without a warning. In Pastry, a node is considered failed when communication between a node and its immediate neighbours is no longer possible.

In order to replace a failed (or departed) node in the leaf set, its neighbour in the nodeId space contacts live node with the largest index on the side of the failed node, and requests that node's leaf table. The received leaf set L' partly overlaps the present node's leaf set L, and it contains nodes with nearby ids not presently in L. From this set of new nodes, the appropriate one is chosen to insert into L to replace the failed node. Also it is necessary to verify that this new node is actually alive by contacting it. Therefore, this procedure guarantees leaf set reparation for each of the nodes that had failed node in its leaf set. [20]

To replace a failed node in the routing table, for example routing table entry R_l^d, a node contacts first one of the nodes of the same row, i.e. R_l^i where $i \neq d$, and requests that node's entry for R_l^d. In case that none of the nodes in row l have a pointer to a live node with the appropriate prefix, the node casts a wider net by contacting an entry R^i_{l+1} where $i \neq d$. This approach guarantees that an appropriate node will eventually be found, if one exists. [20]

In case that one of the nodes from the neighbourhood set needs to be replaced, the node asks other members of the neighbourhood set for their neighbourhood tables, checks the distance of each of the newly identified nodes, and updates its own neighbourhood set accordingly.

2.7 Chord

Chord is a distributed lookup protocol that uses one-dimensional circular key space to provide fast computation of a hash function mapping keys to nodes responsible for them. It uses consistent hashing [22], which ensures, with high probability, that all nodes receive roughly the same number of keys (i.e. balanced load). Also, Chord improves the scalability of consistent hashing by eliminating the condition that every node is familiar with every other node. Therefore, a Chord node needs only a small amount of routing information about other nodes, so in an N-node network, each node maintains information about only O(log N) other nodes, and a lookup requires O(log N) messages. [21]

When a node joins or leaves the network, Chord must update the routing information, which requires $O(\log^2 N)$ messages, and only an O(1/N) fraction of the keys are moved to different location to maintain a balanced load.

The consistent hash function uses a base hash function such as SHA-1 [23] to assign each node and key an m-bit identifier, where node's identifier is chosen by hashing the node's IP address, while a key identifier is produced by hashing the key. It is highly important that the length of identifier m is large enough to minimize the probability of two nodes or keys hashing to the same identifier.

Consistent hashing assigns keys to nodes by ordering identifiers in an identifier circle modulo 2^m. The node responsible for the key k is the first node whose identifier equals or follows (the identifier of) k numerically in the identifier space; that node is called the key's successor, denoted by successor(k). Since a circle of numbers from 0 to $2^m - 1$ represents identifiers, then successor(k) is the first node clockwise from k. Figure 11 shows an identifier circle with m = 3, which has three nodes: 0, 1 and 3. The successor of identifier 1

is node 1, so key 1 would be located at node 1. In the same way, key 2 would be located at node 3, and key 6 at node 0. [21]

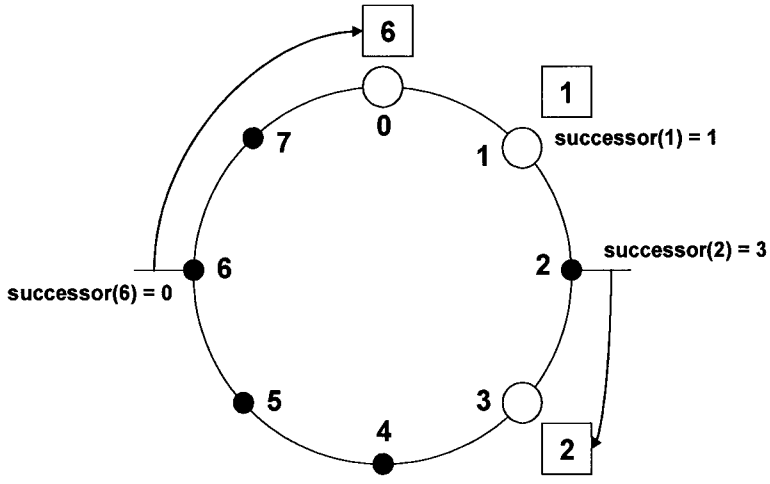


Figure 11.An identifier circle consisting of three nodes 0, 1, and 3

Join/leave process in consistent hashing is designed to allow nodes join and leave the network with minimal disruption. In order to maintain the consistent hashing mapping when a node n joins the network, a number of keys previously assigned to n's successor now become assigned to n. On the other hand, if node n leaves the network, all of its assigned keys are reassigned to n's successor. These are the only changes in assignment of keys to nodes that need to occur. For the example presented above (Figure 11), if a node were to join with identifier 7, it would capture the key with identifier 6 from the node with identifier 0. [21]

As stated previously, in Chord, each node needs only to be aware of its successor node on the circle. However, in order to have efficient routing, i.e. not to require traversing all N nodes to find appropriate mapping, Chord maintains additional routing information. Each node in Chord has a successor list of k nodes that immediately follow it in the key space (used to achieve routing correctness), but on top of that each node maintains a routing table with (at most) m entries, called the finger table (used to achieve routing efficiency). A finger table entry includes both the Chord identifier and the IP address of the relevant node, and the i^{th} entry in the finger table at node n contains the identity of the first node s, also called the i^{th}

finger of node n, that succeeds n by at least 2^{i-1} on identifier circle. Routing efficiency is achieved with nodes in the finger list spaced exponentially around the key space. [21]

Routing in Chord consists of forwarding to the node closest, but not past, the key k. More specifically, if a node n does not know the successor of a key k, then it searches its finger table to find the node j with an ID that is closer than its own to k, and that node j will know more about the identifier circle in the region of k than n does. Therefore, when node n finds node j in its finger table, it asks node j for information on the node that j knows whose ID is closest to k. This process is repeated until n learns about nodes with IDs closest to k. This method requires O(log N) hops. [21]

2.8 Content Addressable Network (CAN)

The Content Addressable Network (CAN) is a distributed decentralized peer-to-peer architecture that is based on a virtual d-dimensional Cartesian coordinate space on a d-torus. This P2P infrastructure provides hash table functionality on Internet-like scale, and it is completely logical and dynamically partitioned among all the nodes in the system such that every node possesses its individual distinct zone within the overall space. An example of a 2-dimenisonal [0, 1] x [0, 1] coordinate space partitioned between 5 CAN nodes is shown in Figure 12. [24][31]

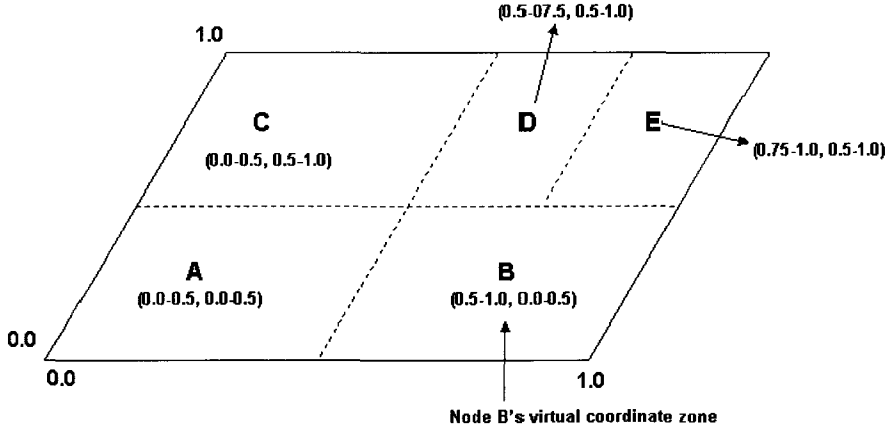


Figure 12.Example 2-D coordinate overlay with 5 nodes

Each node in CAN architecture maintains a routing table that holds the IP address and virtual coordinate zone of each of its neighbours in the coordinate space. A CAN message includes the destination coordinates used by a node to route a message towards its destination using a simple greedy forwarding to the neighbour node that is closest to the destination coordinates. For a d dimensional space partitioned into N equal zones, CAN has a routing performance of $O(d\ N^{1/d})$. Figure 13 gives an example of a routing path for CAN. [24][31]

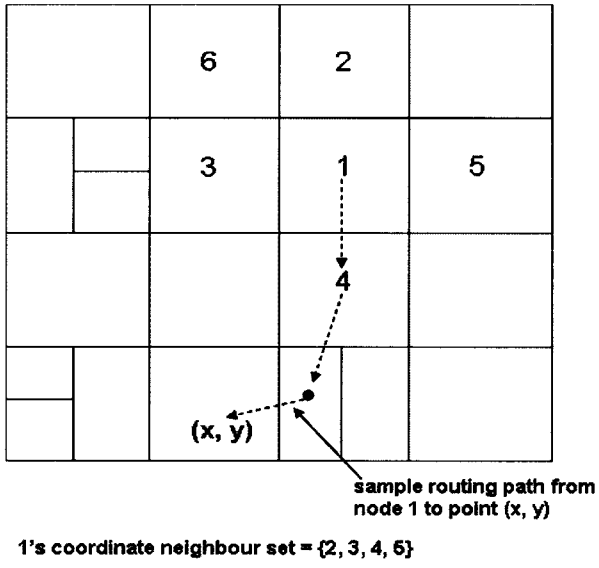


Figure 13.Example routing path

As presented in Figure 13, the virtual coordinate space is used to store (key, value) pair as follows: a key-value pair (K, V) is stored by using a uniform hash function to deterministically map a key K onto a point P in the coordinate space. In order to retrieve an entry corresponding to key K, any node can use the same deterministic hash function to map K onto point P and then retrieve the corresponding value V from the point P. If the point P is not owned by the requesting node or any of its immediate neighbours, the request is routed through the CAN infrastructure until it reaches the node where P is located. Since each node in CAN maintains a list of IP addresses for nodes that hold coordinate zones adjacent to its zone, this list of direct neighbours in the coordinate space is used as a routing table that allows efficient routing between points in this space. [24]

When a new node joins the CAN system, a portion of the coordinate space is allocated by splitting existing node's zone in half; retaining one half for the current node and allocating the other half for the new node. In order to join CAN network, the node looks up in the DNS, which is resolved into IP address of one or more CAN bootstrap nodes (which maintains a partial list of CAN nodes), a CAN domain name to retrieve a bootstrap node's IP address. The bootstrap node provides the IP addresses of several randomly chosen nodes in the system from which the new node chooses a point P and sends a JOIN request destined for point P. This JOIN request message is forwarded by CAN nodes using the CAN routing method until it reaches the node in which zone P is located. When the P's zone is located, the current node in that zone splits it in half and assigns the other half to the new node. More specifically, in a 2-dimensional space, a zone would first be split along X dimension, then Y dimension, and so on. All the key-value, i.e. (K, V), pairs that are located in the half zone to be handed over are also transferred to the new node. At the end, after the zone is obtained, the new node learns the IP addresses of its neighbour set from the previous node in point P, and adds to that previous node itself. [24]

On the other hand, when node leaves the CAN network, an instant takeover algorithm makes sure that one of the neighbour nodes takes over the zone. Also, the node updates its neighbour set to eliminate those nodes that are no longer its neighbours, and each node in the system sends soft-state updates to ensure that all of their neighbours learn about the change and update their own neighbour sets. The total number of nodes in the system does not influence the number of neighbours a node maintains, since it only depends on the coordinate space dimensionality. [24]

2.9 Kademlia

The Kademlia is another P2P decentralized overlay network that supports distributed hash table functionality (DHT). It takes the basic approach of assigning each node a NodeID in the 160-bit key space, and (key, value) pairs are stored on nodes with IDs "close" to the key. Also, it provides a lookup algorithm that locates successively closer nodes to any desired ID, converging to the lookup target in logarithmically many steps.

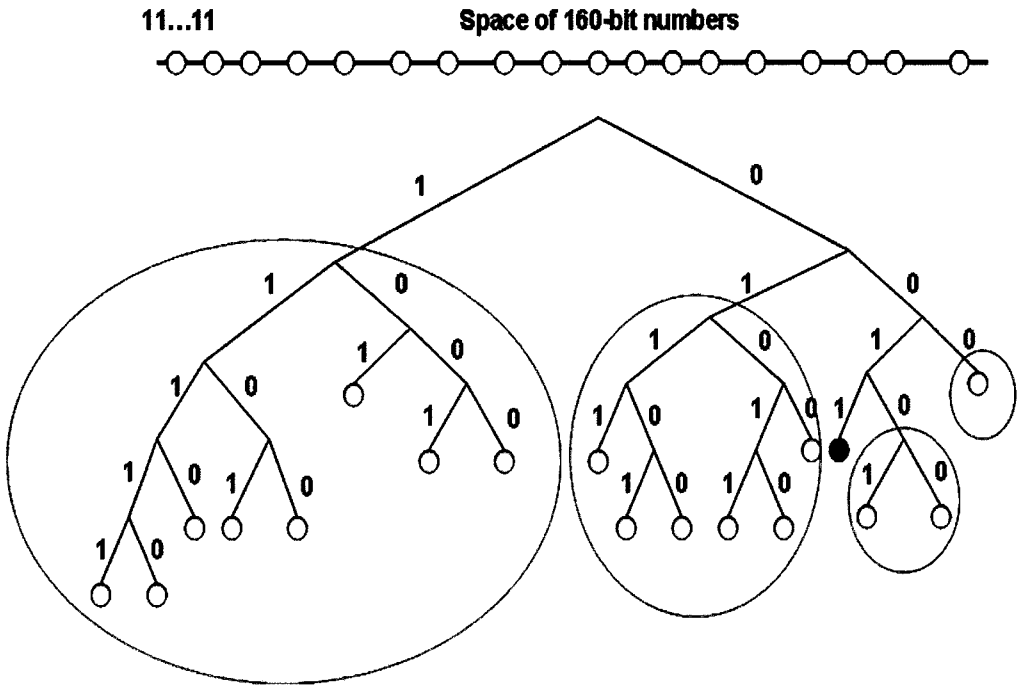


Figure 14.Kademlia binary tree

Each node in Kademlia is treated as a leaf in a binary tree, where node's position is determined by the shortest unique prefix of its ID. Figure 14 gives an example for a node with unique prefix 0011 in an example binary tree. For any particular node, the binary tree is divided into a series of successively lower subtrees that don't contain the node, where the highest subtree consists of the half of the remaining tree not containing the node, and the next subtree consists of the half of the remaining tree not containing the node, and so on. In the example presented in Figure 14 (example for node 0011), the subtrees consist of all nodes with prefixes 1, 01, 000, and 0010 respectively. [25]

In Kademlia any node can locate any other node by its ID because every node knows at least one node in each of its subtrees (if that subtree contains a node). Figure 15 gives an example of NodeID based routing algorithm where node 0011 locates node 1110 by successively querying the best node it knows of to find contacts in lower and lower subtrees. [25]

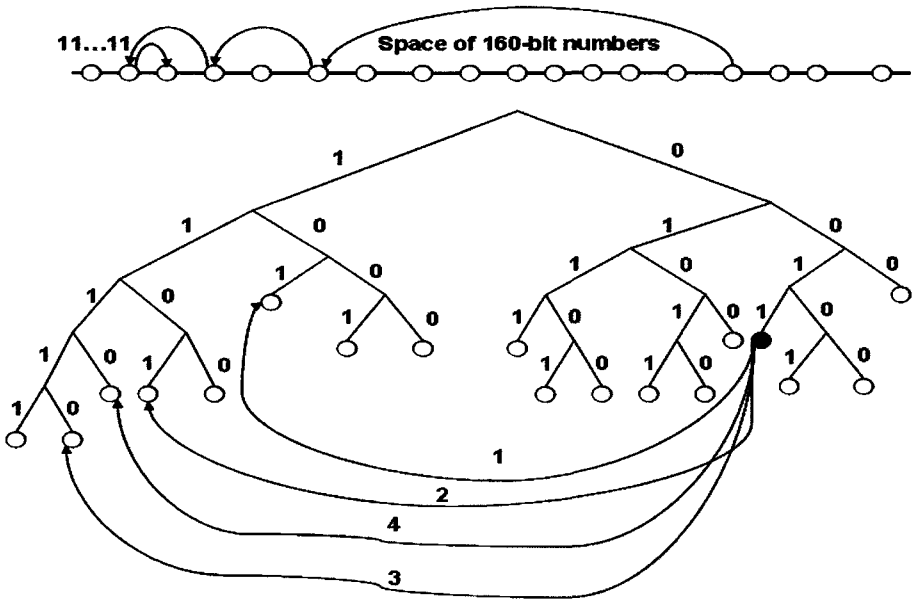


Figure 15.Locating a node in Kademlia binary tree by its ID

In Kademlia every message being transmitted by a node includes its NodeID, permitting the recipient to record the sender's existence. In order to locate (key, value) pairs, Kademlia relies on a notion of distance between two 160-bit identifiers. It defines the distance between these two identifiers as their bitwise exclusive or (XOR). [25]

Similarly to Chord's clockwise circular metric, XOR is unidirectional, i.e. for any given point x and distance $d > 0$, there is exactly one point y such that $d(x, y) = d$. This specific property makes sure that all lookups for the same key converge along the same path, regardless of the originating node, so caching (key, value) pairs along the lookup path alleviates hot spots. [25]

In order to route query messages, every node in Kademlia keeps a list of (IP address, UDP port, NodeID) triples, called *k*-bucket, for nodes of distance between 2^i and 2^{i+1} from itself (where $0 \leq i < 160$). Each *k*-bucket is kept sorted by last time seen, specifically, least recently accessed node is stored at the head, most recently accessed node is stored at the tail of the list. [25]

The Kademlia routing protocol consists of four major actions: [25]

- PING – probes a node to see if it is online.
- STORE – instructs a node to store a (key, value) pair for later retrieval.
- FIND_NODE – takes a 160-bit ID, and returns (IP address, UDP port, NodeID) triples for the *k* nodes it knows that are closest to the target ID.
- FIND_VALUE – behaves similarly to FIND_NODE by returning (IP address, UDP port, NodeID) triples, except for the case when a node received a STORE for the key, then it just returns the stored value.

Node lookup is the most important procedure that Kademlia's node must perform, i.e. it needs to locate the *k*-closest nodes to some given NodeID. The lookup initiator starts by picking x nodes from its closets non-empty *k*-bucket, and then it sends parallel asynchronous FIND_NODE to the x nodes it has chosen. If FIND_NODE fails, i.e. it does not find a node that is any closer than the closets nodes already seen, the initiator resends FIND_NODE to all of the *k* closest nodes it has not already queried. The lookup process terminates when the initiator has queried and received responses from the *k* closest nodes it has seen. [25]

In order to find a (key, value) pair, a node starts by performing a FIND_VALUE lookup to find the *k* nodes with IDs closest to the key. The process terminates immediately as soon as one of the nodes returns the value.

To join Kademlia network, a node *n* must have contact to a node *m* that is already part of the network. In the first step of the join process, node *n* inserts node *m* into the appropriate *k*-bucket, and then performs a lookup for its own NodeID. At the end of the join process, *n* refreshes all *k*-buckets further away than its closets neighbour, and at the same time it populates its own *k*-buckets and inserts itself into other nodes' *k*-bucket as needed. [25]

2.10 Viceroy

The Viceroy P2P decentralized overlay network uses distributed hash table (DHT) to manage distribution of data among changing set of servers and allows clients to contact any server in the network to locate any stored resource by name. Viceroy utilizes consistent hashing [22], to distribute data so that it is balanced across the set of servers and resilient to servers joining and leaving the network. It also maintains an architecture that is an approximate butterfly network [27] (Figure 16 illustrates an ideal Viceroy network), and uses the connected ring of predecessor and successor links for short distances (ideas that were based on Kleingberg [28] and Barriere et al. [29]). On top of predecessor and successor links, each server in Viceroy includes five outgoing links to specific long range contacts. Viceroy's diameter of the overlay is better than CAN and its degree is better than Tapestry, Pastry and Chord. [26]

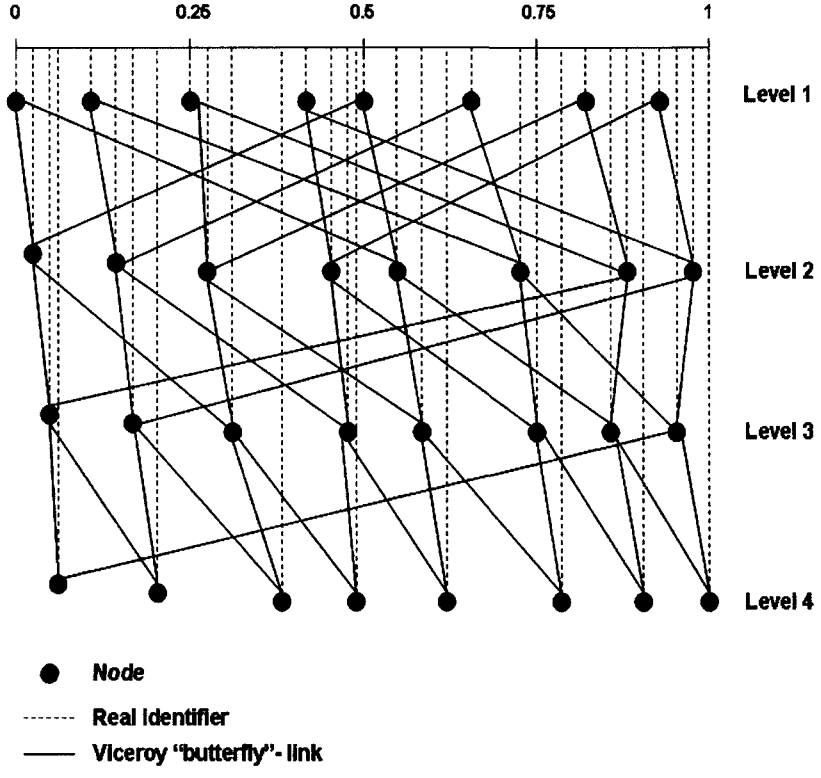


Figure 16.An ideal Viceroy network

When n nodes are operational, each node randomly selects a level in such a way that one of log n levels is selected with nearly equal probability. For a level l node, two edges are connected to nodes at level $l + 1$. Specifically, a down-right edge is added to a long-range contact at level $l + 1$ at a distance about $1/2^l$ away, and a down-left edge at a close distance on the ring to level $l + 1$. Also, an "up" edge to a nearby node at level $l - 1$ is included if $l > 1$. At the end, "level-ring" links are added to the next and previous nodes of the same level l. [26]

Routing is done in three stages. In particular [26],

- The first stage is done by climbing using up connections to a level $l - 1$ node

- Then, in the second stage of the routing process, the down links are used to proceed down the levels of the tree, moving from level l to level $l + 1$, and following either the edge to the close-by down link or the far-away down link, depending weather distance is greater than $1/2^l$ or not. This process is recursively repeated until a node is reached with no down links, which presumably is in the vicinity of the target.

- In the last stage, a vicinity lookup is performed using the ring and level-ring links until the target is reached (which is not necessarily a leaf in the tree)

It has been formalized and proven in [26] that this routing process requires only O(log n), where n is the number of nodes in the network.

Chapter 3

Applying Multidimensional Indexing and Multidimensional Binary Search Tree to MMVEs

3.1 Multidimensional Indexing

Multidimensional indexing is based on the fact that one record can have multiple attributes specified as constituting the primary key set for that record, and that all keys should have the same importance in determining the physical placement of a record in store and consequently in optimization issues. Even though every member of the key set must be defined for a record, it is only required that the whole key set is unique, but not its individual components. More specifically, if we consider indexing of objects at points in three dimensional space, objects can have any one or any two points the same, i.e. they can lie on the same line or the same plane, but no two objects can occupy the same point and consequently all three coordinates cannot be the same (as is the case with MMVE where objects are indexed by their three coordinates in the virtual environment which are used as attributes).

With multidimensional data structures the algorithms for retrieval are significantly more complex than the algorithms for one-dimensional structures. More specifically, with a three attributes key set, it could be required to retrieve all those records for two specific values of the key set but with any value for the other one; or records whose key set attributes values lie within the specific range. Therefore, the development of algorithms that deal with the retrieval of multidimensional records is a complex research subject matter, since they are considerably more difficult than those which determine the placement of a given record in physical storage.

Multidimensional data structures are applied in many real index applications, from CAD, VLSI and geographical databases to multimedia and time series management systems. There are a lot of additional applications of multidimensional data structures, i.e. data mining, indexing multimedia data, indexing of text documents and images, term indexing, XML documents, combinatorial optimization, parallel processing, etc.

In multidimensional indexing the objective is to map the points of the multidimensional space by an indexing scheme mapping each point to a one-dimensional index, i.e. a natural number in the range between 1 and the total number of points in the space. Also, it is highly desirable that this indexing scheme preserves locality, i.e. close by multidimensional points are mapped to close by one-dimensional indexes or vice versa. Space-filling curves have been shown to have that specific property (i.e. preserve locality of multidimensional points mapped to one-dimensional indexes).

In the next section, we present a Z-order space filling curve. Proposed first in 1966 by G.M. Morton [32], the Z-order space filling curve of appropriate order gives a linear ordering to the points in the virtual environment to construct an one-dimensional index to a multidimensional space. Therefore, an object's virtual environment three dimensional (3D) coordinates are mapped to the nearest vertex of the Z-order curve, and the index number of that vertex is then used as the object's key within the overlay. This process, together with a detailed description of the Z-order curve, is described next.

3.2 Z-order Space-Filling Curve

In this section we describe Z-order space-filling curve. Due to its good locality preserving behavior, it is often used in data structures for mapping multidimensional data to one dimension. Z-order space-filling curve maps multidimensional spaces to a compact interval by passing close to every point in the space. Z-order curve has already been applied for several applications, specifically, for indexing regions (e.g. Bank File) and for spatial data (e.g. Orenstein's PROBE Project [33]).

Z-order curve derived index of a point in multidimensional space is assembled simply by cyclically taking a bit from each coordinate of a point, starting from a most significant bit (MSB), and appending it to those taken previously. This process is referred to as "bit interleaving", and it is presented in Figure 17 (an example of bit interleaving for a two-dimensional (2D) point, with coordinates x and y). [34][36]

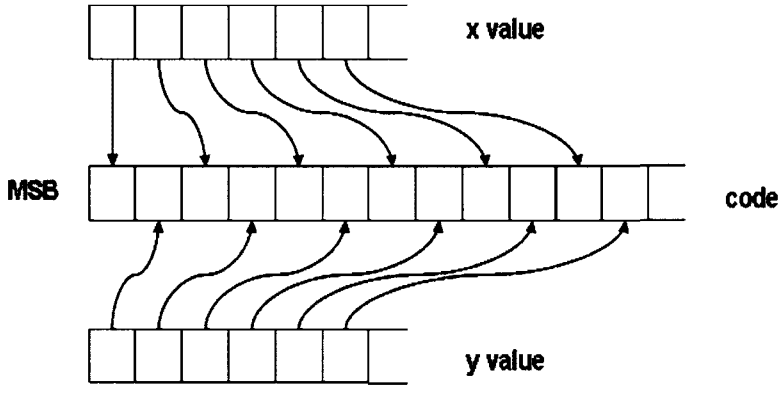


Figure 17.Bit interleaving

Let us look at a 2D point P with bit representation of its coordinates (x, y) as

$$(x_1x_2x_3\ldots x_k,\ y_1y_2y_3\ldots..y_k).$$

The "bit interleaving" process maps P's coordinates to a Z-order derived index of

$$(x_1y_1x_2\ y_2x_3y_3\ldots..x_ky_k),$$

where each x_i is a bit of x-coordinate and each y_i is a bit of y-coordinate. In Figure 18, the Z-order indexes are shown for the two-dimensional case with integer coordinates $0 \leq x \leq 3$ and $0 \leq y \leq 3$ (shown both in decimal and binary representation). Binary Z-order index values are produced by interleaving the binary coordinates as shown in Figure 18. The Z-shaped curve is produced by connecting these Z-order index values in their numerical order.

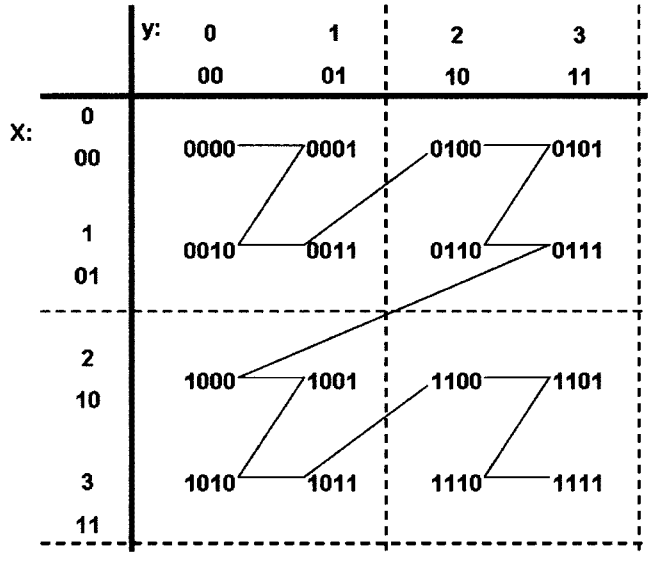


Figure 18. Z-order index calculation example

The mapping of the first order curve consists of a single rotated Z shape, and subsequent orders consist of Z's that are rotated and translated. These mapping of the Z-order curves are illustrated in Figure 19, which shows the first three iterations (orders) of the curve with vertexes in two dimensions. For the purpose of clarity, explanations and illustrations in this chapter are presented in the two dimensional paradigm, but it can easily be applied to the n dimensional case. [35]

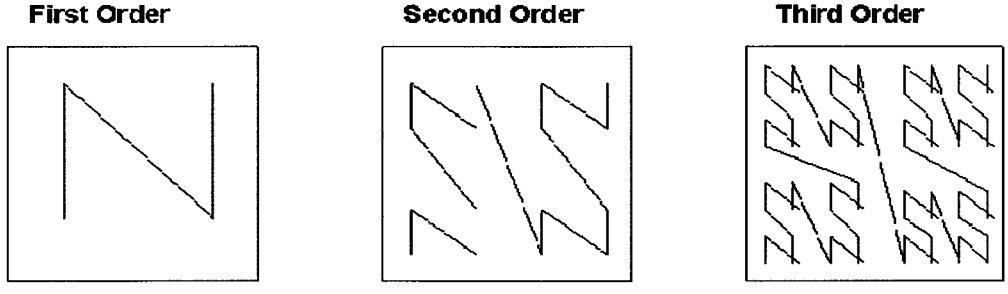


Figure 19.Z-order space-filling curves in two-dimensions

As stated previously, the main idea behind the bitwise interlacing scheme is that nearby multidimensional points should also be close to each other after mapping to one dimension. The Z-order space filling curve presented here tries to group these points together, a feature

desirable when searching for compact ranges. Since KD-trees (see chapter 4) have been shown well suited for range searching and they also group the data points together alternating the discriminating keys, it is obvious that there is a relationship between KD-trees and bitwise interlacing process, and that KD-trees are a suitable data structure for presenting one-dimensional order of multidimensional data points. With this in mind, we propose using KD-trees, as explained next.

3.3 Multidimensional Binary Search Tree (KD-tree)

Next we present a particular data structure, the multidimensional binary search tree, i.e. the KD-tree. We also explore KD-tree suitability for representation of multidimensional indexing using Z-order space filling curve.

The multidimensional binary search tree (KD-tree) is a space partitioning data structure for points in a k-dimensional space. KD-tree is a natural extension of the binary search tree to k-dimensions, and it is designed to handle the case of a single record having multiple keys (dimensions). More specifically, KD-tree is a data structure that is useful for any application that requires searches involving multidimensional search key; i.e. range searches and nearest neighbour searches. It is a special case of the Binary Space Partitioning (BSP) tree.

The remainder of this chapter gives background information on one-dimensional binary search trees, and detailed description of multidimensional binary search trees and its "search queries" properties. We also give detailed description of our architecture, based on Z-order space filling curve and KD-tree, which can effectively handle issues related to objects in 3D virtual environment where locations are described in terms of 3D coordinates.

3.4 One-dimensional Binary Search Trees

In this section we give a brief description of binary search trees. The main characteristic of a binary search tree for any node x are as follows (as shown in Figure 20): [37]

- The key values in the left subtree of x are all less than the key value of x
- The key values in the right subtree of x are all greater than the key value of x
- Both the left and the right subtrees must also be binary search trees

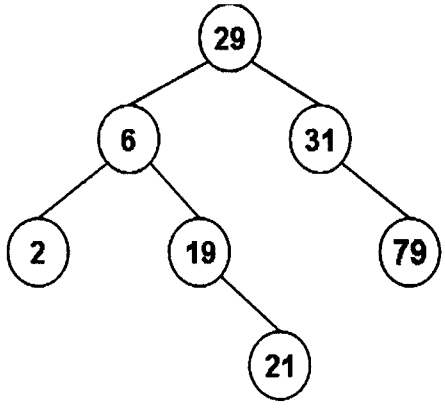


Figure 20. One-dimensional binary search tree

Binary search trees are a multi-task data structure, since they can "store" the records of a given set, they impose a "partition" on the data space, and they provide a directory that allows quick location of a new point in the partition by making a logarithmic number of comparisons. The major advantage of binary search trees is that the related sorting algorithms and search algorithms such as "in order traversal" can be very efficient. Also, they are a fundamental data structure used to construct more abstract data structures such as sets, multi-sets, and associative arrays. [37]

3.5 KD-trees

The difference between standard binary search tree and KD-tree is that a standard binary search tree makes its insertion decision based on one key field, whereas KD-tree makes its insertion decision based on multiple keys. Specifically, if we assume that each record has n keys, K_1, K_2,..., K_n, on the first level of the KD-tree (i.e. root of the tree) we decide to go left

or right when inserting a new record comparing the first key (K_1) of the new record with the value stored at the root of the KD-tree. At the second level of the tree we use the second key (K_2) and compare it with the value stored at the appropriate internal node at second level, based on the previous left/right decision at the root node, and so on until we reach the n^{th} level. At the (n + 1) level of the tree we loop and use the first key again until we reach the appropriate leaf node where the record should be stored. This concept is illustrated in Figure 21, where 2-dimensional tree is presented with records containing two keys: name (K_1) and age (K_2). In the tree presented below, each record in the left subtree of the root has a name field less than the root's, and equally each record in the right subtree has a greater name filed. Also, same method is applied on the second level with age, where right subtrees have greater age values. [37]

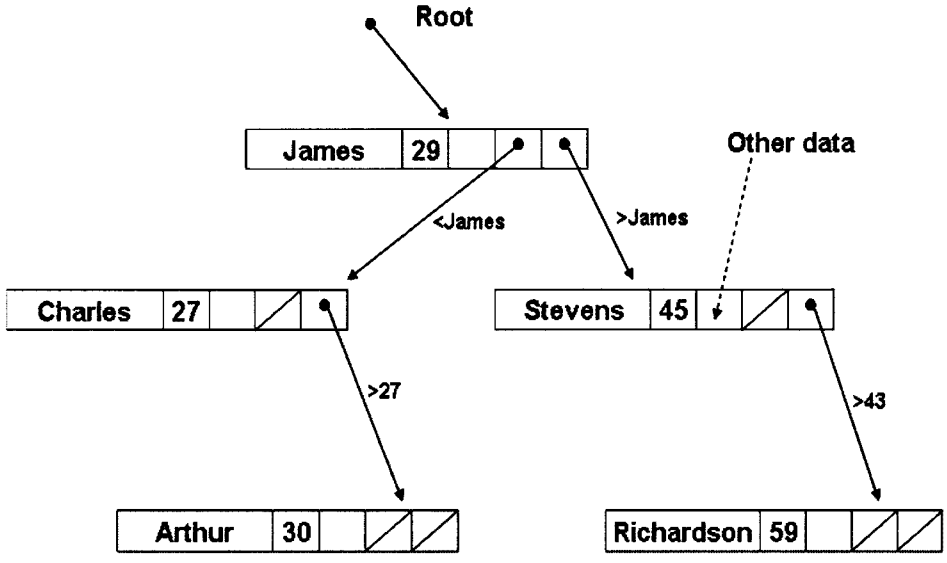


Figure 21.A 2-dimenisonal tree: K_1 is name and K_2 is age

Therefore, a KD-tree is formally defined as a binary tree where each node contains k keys, some data fields, left and right pointers, and a discriminator value which is an integer between 1 and k (inclusive). The essential characteristic of KD-trees is that for any node x which is an i-discriminator, all nodes in the left subtree of x have K_i values less than x's K_i value, and all nodes in the right subtree have greater K_i value. As described previously, to insert a new record we start at root and go down the tree by comparing at each node visited one of the new record's keys with one of the keys of that node (i.e. the one specified by the

discriminator), a process that requires approximately 1.386 lg N comparisons, on average (shown by Bentley [38]). Conversely, in order to perform N insertions, the expected cost is O (N lg N). [37]

The same process that is described above is applied for organizing points in a k-dimensional space, where each point is considered as a record and its coordinates are treated as keys. For example, if we have a 2-dimensional (2D) point P with coordinates (X, Y), the values of X and Y are used as keys K_1 and K_2 for a comparison with the appropriate values of the internal nodes for the KD-tree data structure that partitions the 2D space in which that point is located. Next we will show how a KD-tree is constructed for a set of points in 2 dimensions (the same method applies for k-dimensional points too, where k = 1.....n).

Let consider a set P of n points in 2 dimensions, where each point $p_i \in P$ is specified by its coordinates (x_i, y_i). The plan is to split the points based on value of x-coordinate first, then on y-coordinate, then again on x-coordinate, then again on y-coordinate, and so on, until the number of the points in the subset to store in a sub-tree reaches some minimal value m (in this case m = 1). When we reach this minimal value m, a leaf node is created to store all the points in this subset. Therefore, the process is as follows:

1. First, at the root, we split P with a vertical line into two subsets of equal size (based on x-coordinate of the points in P), where the subset of points to the left of the splitting line or on the splitting line is stored in the left sub-tree (P_{left}) and the subset on the right of the splitting line is stored in the right sub-tree (P_{right}).

2. Next, we split P_{left} (the left child of the root) into two subsets with a horizontal line (based on y-coordinate of points in P_{left}), where points below or on this horizontal line are stored in the left sub-tree of the P_{left}, and points above this horizontal line are stored in the right sub-tree of the P_{left}. The same process is repeated for points in P_{right} but with a horizontal line that is based on y-coordinate of points in P_{right}.

3. We repeat steps 1 and 2 until we reach a subset of points with a specified minimum size of m. See Figures 22 and 23 for an example when m = 1.

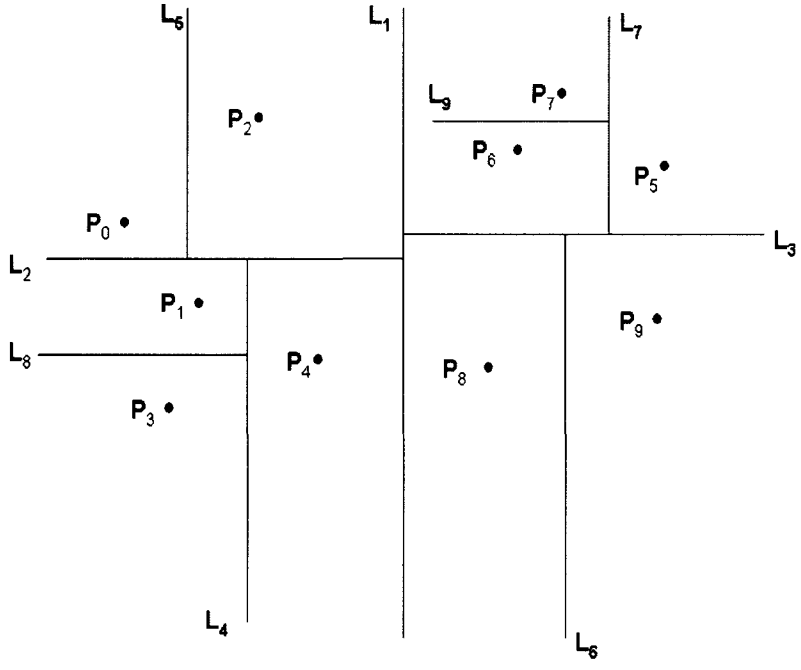


Figure 22.Space partitioning

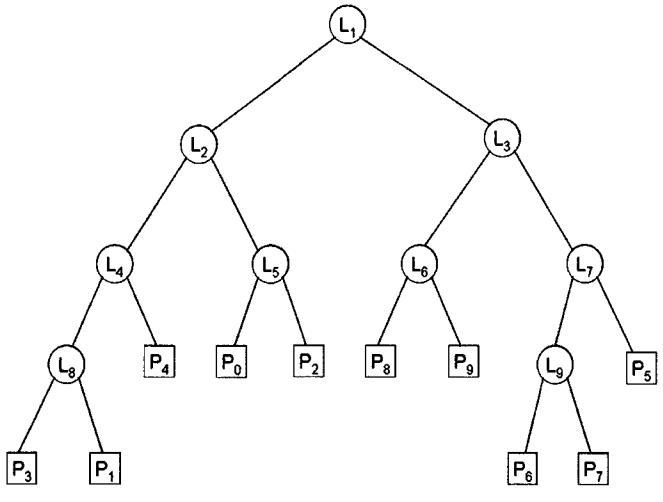


Figure 23. KD-tree construction based on space partitioning

In order to have balanced KD-tree, the splitting line that divides subsets at each step of the process is based on the median value of the points for the appropriate coordinate, i.e. at each internal node of the KD-tree, the splitting line passes through the median x-coordinate at the root level, then the median y-coordinate at the first level, then again the median x-coordinate at second level, and so on. In general, for 2 dimensional coordinates, we use the median x-coordinate if the depth of the KD-tree is even or the median y-coordinate if the depth is odd.

Each node in KD-tree (both internal and leaf node) corresponds to a rectangular area of the plane, where some of these areas will be unbounded. More specifically, the root node corresponds to the entire plane, and the points in the left subtree of the root correspond to the left half-plane, and ,conversely, points in the right subtree correspond to the right half-plane. With this in mind, i.e. that each node of the KD-tree stores the bounding box of its rectangular region, KD-tree construction algorithm is shown in pseudocode form below. First, let us define some notation. [40]

P: a set of points
depth: KD-tree depth
Bbox: bounding box (as described above)
m: minimum value of the subset of points to store in the leaf node during the process of splitting the points (as described above)

```
Build _KdTree (P, depth, Bbox, m) {
        if P contains m or fewer points
        then return a leaf node v storing these points and the node rectangle Bbox
        else if depth is even
                then Split P into two subsets with a vertical line l through
                        the median x-coordinate of the points in P.
                        Let P1 be the set of points to the left of or on l,
                        and let P2 be the set of points to the right of l.
                        Let Bbox1 be the part of Bbox to the left of l,
                        and let Bbox2 be the part of Bbox to the right of l.
```

else Split P into two subsets with a horizontal line l through
the median y-coordinate of the points in P.
Let P_1 be the set of points below or on l,
and let P_2 be the set of points above l.
Let $Bbox_1$ be the part of Bbox below l,
and let $Bbox_2$ be the part of Bbox above l.

$v_{left} \leftarrow$ Build_KdTree (P_1, depth + 1, $Bbox_1$, m)

$v_{right} \leftarrow$ Build_KdTree (P_2, depth + 1, $Bbox_2$, m)

Create a node v storing l and the node rectangle Bbox,

make v_{left} the left child of v, and

make v_{right} the right child of v.

return v

}

Now that KD-trees construction is described in details, we present different searching algorithms for KD-trees.

3.6 Searching in KD-trees

In this section different algorithms for searching KD-trees are presented. Each of these algorithms is appropriate to answer a specific type of query. The four most commonly discussed types of searches are described in more details, i.e. exact match queries, partial match queries, range queries and best match queries.

3.6.1 Exact Match Queries

The exact match query is the simplest type of query in a KD-tree of k-key records. The process is quite simple and it is similar to the insertion algorithm, i.e. it proceeds down the tree, going left or right by comparing the desired record's key to the discriminator in that node. At the end of the process, the node with the required record is either found on the way

down or the process terminates unsuccessfully by "falling out" of the tree (i.e. it reaches one of the leaf nodes without finding a matching record). [37]

In the worst case scenario, with a perfectly balanced KD-tree, an exact match search requires O(lg N) comparisons. Conversely, for randomly built KD-trees, the average case also takes O(lg N) comparisons. [37]

3.6.2 Partial Match Queries

Partial match query is a bit more complicated type of query since it searches for all records where only certain number of keys needs to be matched. A good example of such a query might occur in employee database, specifically, report all employees with pensionable years of service = 10 and occupational category = 'manager', ignoring all other keys in the records. Usually, with partial match queries, we specify values for t of the possible k keys and ask for all records that have those t values, ignoring the other (k – t) key values. [37]

Partial match query search in KD-trees starts by visiting the root of the KD-tree, and is executed as follows: [37]

- When a node of a KD-tree that discriminates by j-value is visited, we check to see if the value of the j-th key is specified in the query.

- If the value of the j-th key is specified in the query, then only one of the node's children needs to be visited (child that needs to be visited is determined by comparing the desired K_j with that node's K_j value).

- If the value of the j-th key is not specified in the query, then both children must be searched recursively.

It has been shown (Bentley [38]) that if t of k keys are specified, to do a partial match query in a file of N records, it would need approximately $tN^{1-t/k}$ to be examined during the partial

match search. For example, if five out of six keys are specified in a partial match search of one million records, then only approximately 50 records will be examined during the execution of the partial match query.

3.6.3 Range Queries

In this section we describe range querying in KD-trees, where a range of values for each of the k keys is specified, and all the records that have every key in the proper range are reported back as a range query result. More specifically, we might be interested in querying an employee database to find all employees with salary between $30,000 and $50,000, and age between 25 and 45, with at least 5 years of work experience.

Range querying is a common problem that arises in many different applications, and it is easily solvable when records are stored in a KD-tree. Range searching algorithm takes an approach that is similar to the partial match searching algorithm, and it is executed as follows: [37]

- When a node that is a j-discriminator is visited, we compare the j-value of that node to the j-range of the query

- If the range is bellow the j-value of that node, the search continues on the left child

- If the range is above the j-value of that node, the search continues on the right child

It has been shown that a worst case performance of range queries in KD-trees is never more than $O(N^{1-1/k} + F)$, where F is the number of points found in the range. However, the average case for range searching in KD-trees is considerably better, with a performance of $O(\lg N + F)$. It should be noted that it is very difficult to analyze the exact performance of range searching, since it is highly dependant on the "shape" of the particular query, but empirical evidence have shown that KD-trees are very efficient when it comes to range searching. [37]

3.6.4 Best Match Queries

In some applications when an exact match does not exist, the user settles for the most similar item to the originally requested item, usually called "the best match" or the "nearest neighbour". For example, user might search for a book that examines five specific subjects, but if one does not exist, user must settle for one that examines only three out of the requested five.

It has been shown [39] that KD-tree data structure provides an efficient mechanism to answer such best match queries. Full detailed description of this recursive algorithm that depends on choosing the discriminators in a sophisticated fashion is out of the scope of this report and it can be found in [39]. Analysis of the worst case scenario for the best match search in KD-trees has shown that the cost of searching will average to at most $O((\lg N)^k)$.

Chapter 4

VERA: Virtual Environment Routing Algorithm

4.1 Introduction

In this section we present VERA, Virtual Environment Routing Algorithm, based on the proposed multidimensional indexing using Z-order space filling curve and its representation by a KD-tree. Specifically, we propose a proximity based architecture that uses nodes' location to assign unique identifier (NodeID) to allow a faster P2P service in large scale networks (in this case a MMVEs). The ID corresponds to the square cell in which a given node is positioned in a grid partitioned 3-dimensional space. More specifically, the nodes are indexed according to their positions as if they compose distributed database, where nodes are considered as record indexed with a number of keys (in this case three keys, i.e. their 3-dimesional coordinates).

First we show how Z-order curve is used to map 3-dimensional points to one-dimensional indexes, and how these points are stored in proper order (as mapped by Z-order space filling curve) in the KD-tree data structure. Next we show how routing tables are constructed based on the proposed architecture, i.e. multidimensional indexing using Z-order curve and its representation by a KD-tree. At the end we present how the information available in each node's routing table is used to perform efficient routing between points in 3-dimensional virtual environment.

4.2 Indexing with Z-order Curve

As indicated previously, locality sensitive routing is accomplished by conceptualizing a Z-order curve of order k as a KD-tree structure. Each node's routing table is constructed based

on bit-interleaving process (i.e. multidimensional indexing using Z-order space filling curve) and space partitioning with KD-tree data structure.

Therefore, let us look at an example for three-dimensional points with 2 bit coordinates (for simplicity purposes we use points with 2 bit coordinates). For this specific scenario, we need to use a third order Z curve to do space partitioning, since we are going to have 6 bit ordinal numbers as indexes. So, in this specific case, KD-tree space partitioning structure for representation of a multidimensional points indexed by Z-order space filling curve is illustrated in Figure 24.

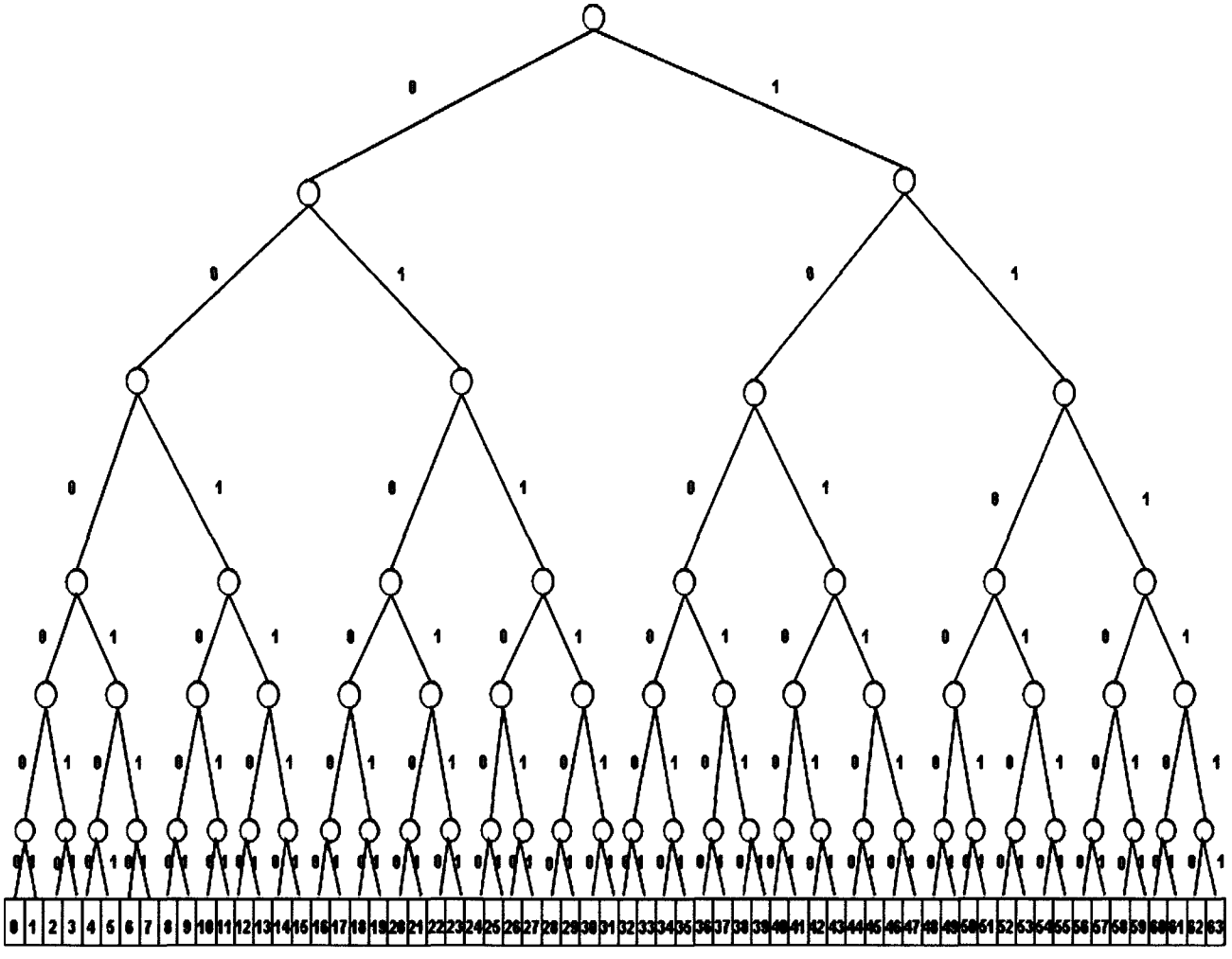


Figure 24. KD-tree for a third order Z-curve

The KD-tree is traversed by taking a bit representation of a one dimensional index (derived from a 3-dimensional coordinate through the "bit- interleaving" process, as described previously) and comparing the bits in a specific order, i.e. we use the most significant bit (MSB) at root of the tree to decide if we should go to the left or the right sub-tree. If MSB is equal to 0 we go into the left sub-tree, otherwise we go into the right sub-tree. Next, at the second level we use second MSB to traverse the sub-tree, then third, and so on until we reach the last bit. At that point, we have reached the leaf node of the tree where this point should be stored in proper order as mapped by a Z-order curve.

Let us look at an example where we have the following set of points: P_1 (0, 1, 3), P_2 (1, 0, 2), P_3 (2, 0, 0), P_4 (1, 2, 1), P_5 (3, 3, 2), P_6 (3, 0, 0) and P7 (2, 1, 3). These multidimensional points are assigned one-dimensional index by bit-interleaving process (Z-order space filling curve) as follows:

P_1 (0, 1, 3) → (00, 01, 11) → (001011) → 11

P_2 (1, 0, 2) → (01, 00, 10) → (001100) → 12

P_3 (2, 0, 0) → (10, 00, 00) → (100000) → 32

P_4 (1, 2, 1) → (01, 10, 01) → (010101) → 21

P_5 (3, 3, 2) → (11, 11, 10) → (111110) → 62

P_6 (3, 0, 0) → (11, 00, 00) → (100100) → 36

P_7 (2, 1, 3) → (10, 01, 11) → (101011) → 43

Next, each of these points is stored, in proper order as mapped by Z-order curve, in the KD-tree as described above. This is shown in Figure 25.

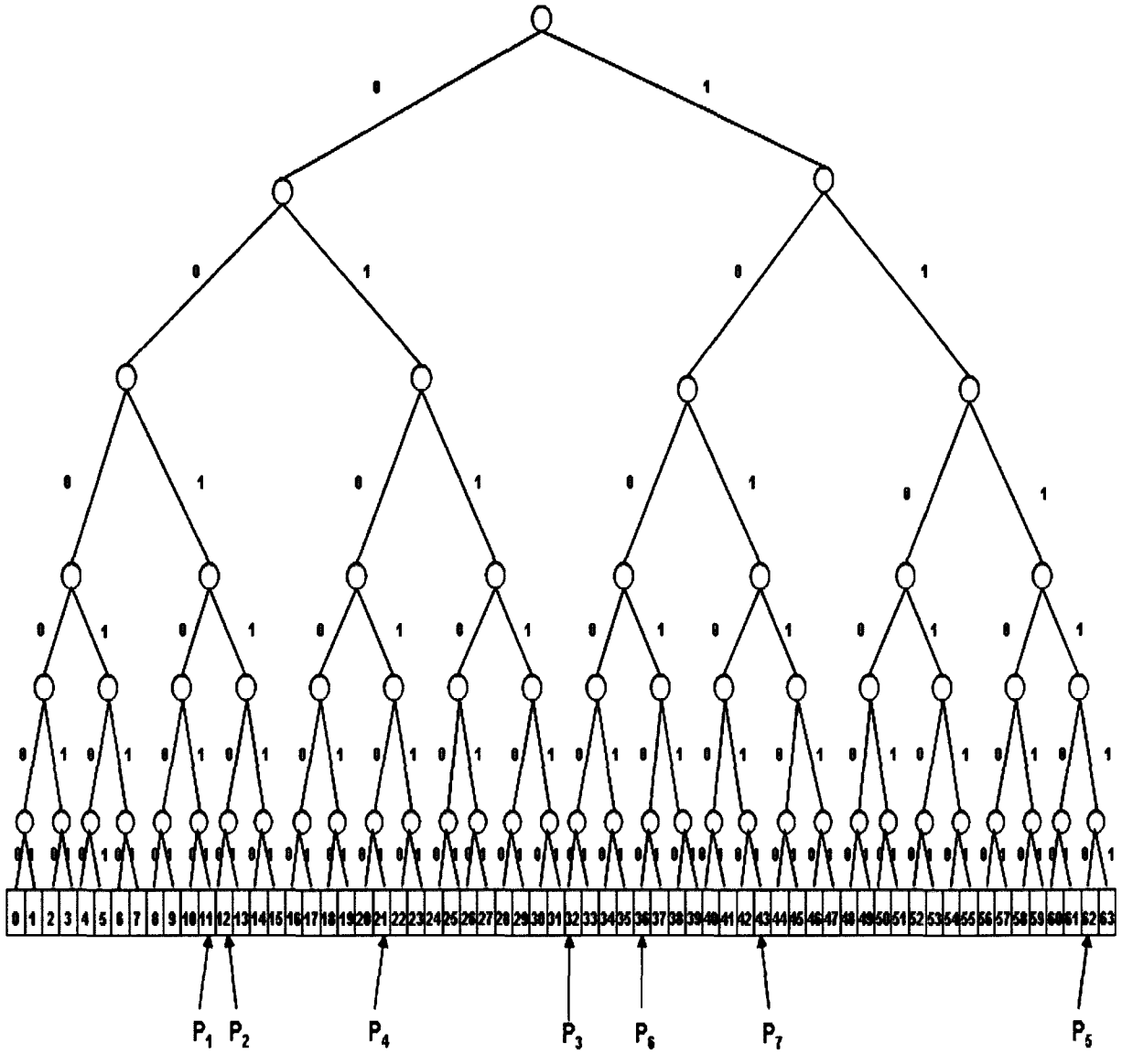


Figure 25. KD-tree storage of points indexed by Z-order curve mapping

It should be noted that bit-interleaving process exactly corresponds to the KD-tree discrimination process, where at each level of the tree different key is used to decide if new record should be inserted in the left or the right sub-tree. In this specific case, the keys are bit representation of coordinates for 3-dimensional points, and at each level of the tree bit representation of different coordinate is used, starting with x-coordinate.

In practice we are interested in higher order curves, since we need to map points whose coordinates have considerably higher number of bits. So, for example, if we had 3-dimensional points with 16 bit coordinates, we would need to use a 24^{th} order Z space filling curve and a tree with height 48, since we would have multidimensional indexes with 48 bits.

Now that we showed how 3-dimensional points are mapped to one-dimensional indexes with Z-order space filling curve, and how these points are stored in the KD-tree data structure, we can present how each node's routing table is constructed.

4.3 Routing Table Construction

Each node maintains a routing table organized in d*b rows with four entries each , where d is the number of VE dimensions (in our case it is three) and b is the number of bits used for coordinates. For example, if we have a 3-dimensional 4 bit coordinates, i.e. (0011, 1011, 1110), we would have nodes that maintain a routing table with 12 rows (since each index would be 12 bits long).

Each entry in the routing table contains the IP address of one of potentially many nodes with a specific position in the KD-tree with respect to the owner of the routing table. More specifically, each level of a node's routing table stores IP addresses of up to four nodes whose index has the same number of matching bits that correspond to that specific level of the routing table, i.e. level 0 node stores an IP addresses of nodes with an index whose MSB is not the same as its own, level 1 stores an IP addresses of nodes with an index whose MSB is the same but second MSB differs, level 2 stores nodes whose index has the first 2 MSB matching but the 3^{rd} bit is different, and so on.

Let us look at an example where we have a set of 3-dimensional points with 2-bit coordinates. Let us assume that we have a set of these points as shown in the KD-tree presented below (Figure 26).

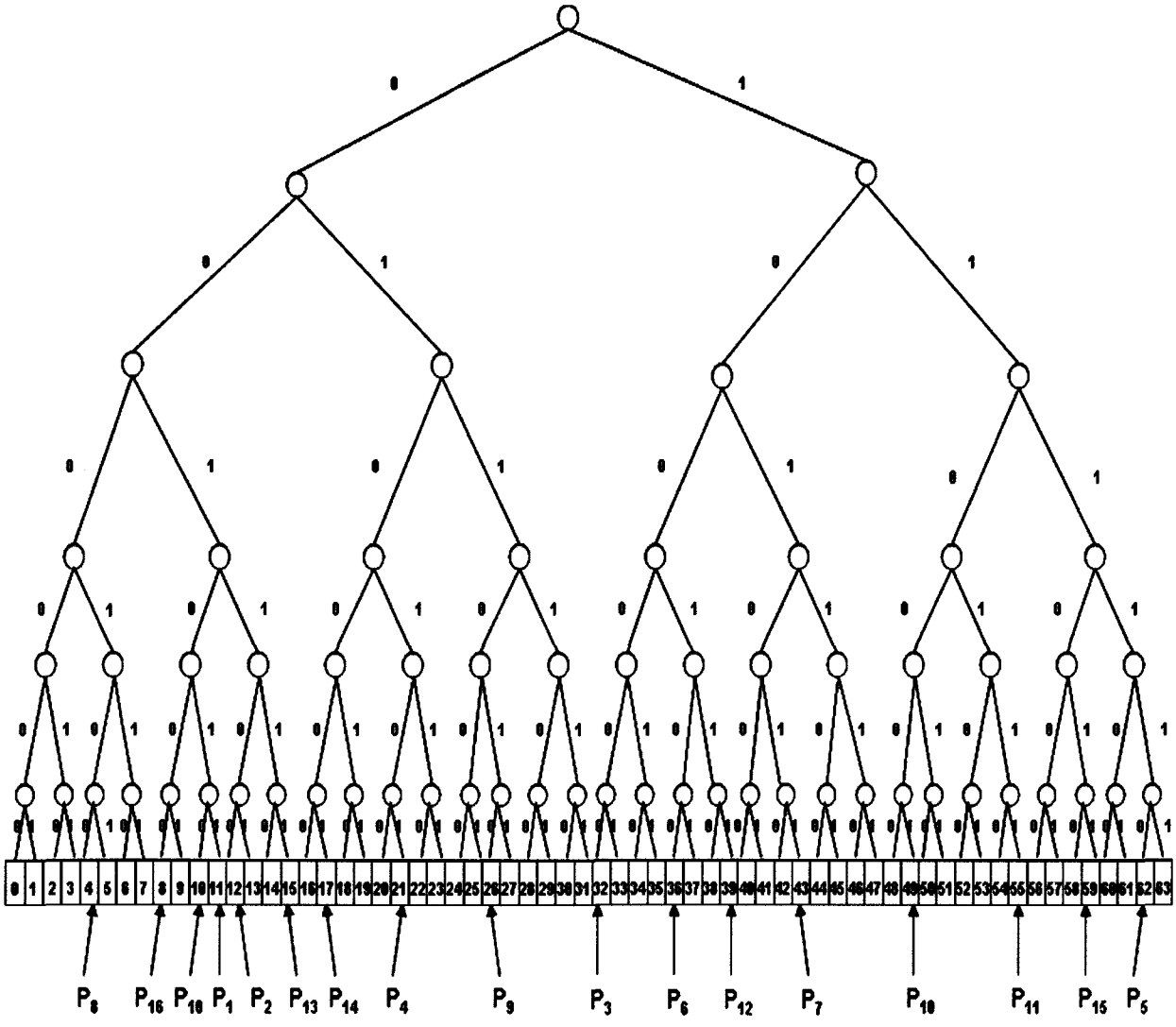


Figure 26. KD-tree example for routing table construction

In Table 1 an example routing table for the node with a NodeID (index) 11 (i.e. point P_1 in Figure 26) is shown. As described previously, the nodes in row 0 have NodeIDs with zero common bits, starting from the MSB, with a node with NodeID 11; nodes in row 1 have 1 bit matching, nodes in row 2 have 2 bits matching, etc. It should be noted that the last couple of rows will have less entries than the upper part of the routing table, since there are more nodes that can have an index with only none or first few bits matching, and only few nodes that have indexes with almost all bits the same (starting from the MSB).

Table 1.Routing table for node with NodeID 11

	0	1	2	3
0	32	36	43	-
1	17	21	26	-
2	4	-	-	-
3	12	15	-	-
4	8	-	-	-
5	10	-	-	-

4.4 Routing Algorithm

In this section we describe VERA that uses each node's routing table to perform an efficient message distribution among nodes. When node A receives a packet destined to node B, the routing algorithm (VERA), based on the structure presented in the previous section, is performed in the following steps.

1. First we assign Start = index (A) and End = index (B)
2. Steps a) to c) are repeated until Start = End, so if Start ≠ End then:

    a. Take the binary representation of indices for Start and End, and check how many bits are equal starting from the MSB
    b. Let us assume that the number of equal bits is I; then we go to row I in the routing table of node Start, and check which of the entries in row I in the routing table has the value closest to the value of End.

    c. Now we set Start value to index of the node identified in step b), and forward a packet to that node

3. If step 3 is reached, the match was found and a packet has reached its destination.

It should be noted that the algorithm presented above is an exact match algorithm, i.e. it assumes that the destination node exists in the MMVE, but it can be easily modified to handle the cases where the destination node does not exist. In that case, the packet is sent to a node with index (i.e. 3D coordinates) that is the closest to original destination point.

Let us look an example using the set of nodes as described in Figure 26. We assume that node with NodeID 11, receives a packet destined to node with NodeID 49. After it received a packet destined to node with NodeID 49, node with NodeID 11 compares its ID with destination ID. Since they do not share any common bits (starting with the MSB), it goes to the row 0 of its routing table, i.e. the first row in its routing table (routing table shown in Table 1), and checks if any of the entries in the first row of the routing table have an ID equal to the destination ID. Given that none of these nodes have NodeID equal to the destination ID, node with NodeID 11 forwards the packet to the node in the row 0 of its routing table with the NodeID that is closets to the destination ID. Therefore, the packet is forwarded to node with NodeID 43.

Next, node with a NodeID 43 repeats the same process, i.e. it compares its ID with destination ID, and derives that their first bit is the same. Therefore, it goes to its routing table's row 1, i.e. the second row, and checks if any of the nodes stored in row 1 have an ID equal to destination ID (routing table for a node with NodeID 43 is shown in Table 2). It locates the node with NodeID 49 in column 0, and forwards the packet to that node. Hence, the process was completed in two routing hops

Table 2. Routing table for node with NodeID 43

	0	**1**	**2**	**3**
0	4	8	10	11
1	49	55	59	62
2	32	36	39	-
3	-	-	-	-
4	-	-	-	-
5	-	-	-	-

4.5 Range Search

In MMVEs very often communication must only occur within subsets of distributed entities, i.e. certain area of interest. In order to have good performance and scalability, it is necessary to successfully determine these subsets, often referred to as interest management, from a globe of many participants. These interest management areas can be roughly classified as either neighbour based or region based. In neighbour based interest management, peers maintain a list of nearby entities with which to communicate, whereas in region based interest management updates are performed through queries on regions.

In this section we present a range search process that is based on the proposed architecture, i.e. Z-order curve and KD-tree. Since this architecture is based on object's actual location in the virtual environment, i.e. 3D coordinates, our approach can be used for both neighbourhood based search, i.e. location of the closest nodes, or region based search (since it is based on actual nodes' location in the network).

Our range search algorithm is based on the binary representation of NodeIDs. Specifically, it is based on the search "prefix", i.e. it searches for all the nodes with NodeIDs starting with a specific "prefix". The length of the "prefix" searched varies depending from the interest management area. For example, if a node wants to find out about all the nodes that are in its immediate neighbourhood, it queries for all the nodes with a "prefix" as close to its own, i.e. if we have a node with 6 bits long NodeID 100110, it would search for all the nodes with a prefix 100. On the other hand, if a node wants to search for wider interest area, it would just query on a shorter prefix (for example all nodes with prefix 1).

Since our architecture and NodeIDs assignment is based on nodes' actual network location, this querying method guarantees that it will return all the current nodes that are in a specific neighbourhood. Therefore, the pseudo-code for the range search based on our architecture is presented next. We begin by defining some notation.

prefix- a list of bits representing the prefix of queried nodes

KD- a KD tree

prefix.first() – returns the first bit in the prefix, i.e. the MSB

prefix.next() – returns prefix without MSB

left_subtree(KD) – returns left sub-tree

right_subtree(KD) – returns right sub-tree

```
range_search(prefix, KD){
    if prefix = = null
        return all nodes stored in the leafs of the KD;
    else
        if first.prefix() = = 0{
            prefix = prefix.next();
            range_search(prefix, left_subtree(KD));
        }
        else{
            prefix = prefix.next();
            range_search(prefix, right_subtree(KD));
        }
}
```

Next, we present an example for range search algorithm. Let us assume that we have a set of nodes as presented in Figure 27, and that a search query with a prefix "0010" is submitted.

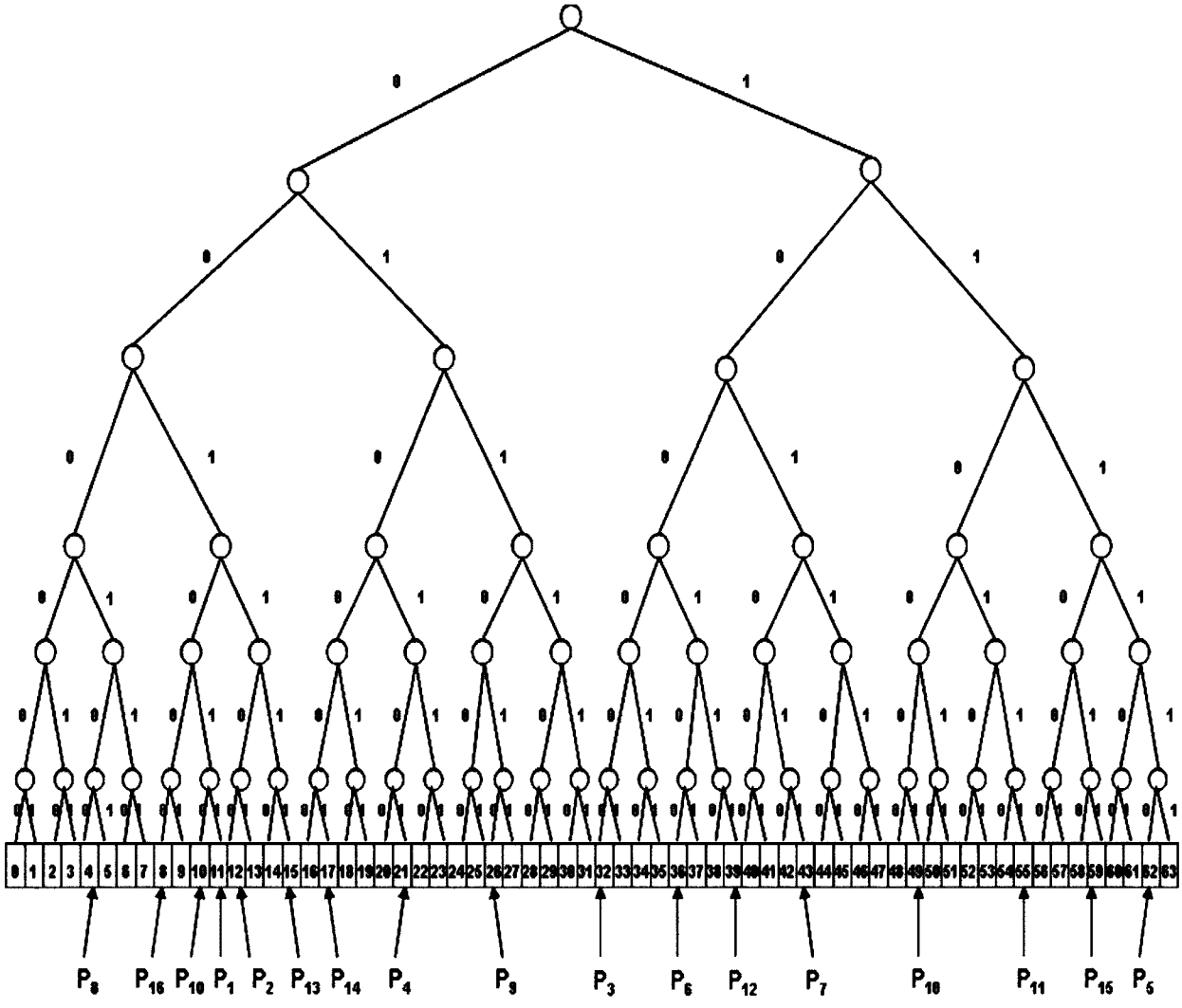


Figure 27. Set of nodes for range search

The query returns nodes with NodeIds 8, 10 and 11, since these nodes have NodeIDs that start with the queried prefix. The querying process is presented in Figure 28.

This querying process presented in this section can be especially useful for new nodes when they join the network, since they can query for nodes in their neighbourhood, i.e. that have a NodeID close to their NodeID. Once they have identified them, they can contact them to obtain their routing table information and they can use this information to organize their own routing tables, since they have common NodeIDs. Also, this process can be used when a node changes its position due to its movement in the virtual environment, i.e. to obtain the

information about its new neighbourhood. These network dynamics are explored in greater detail in the next chapter.

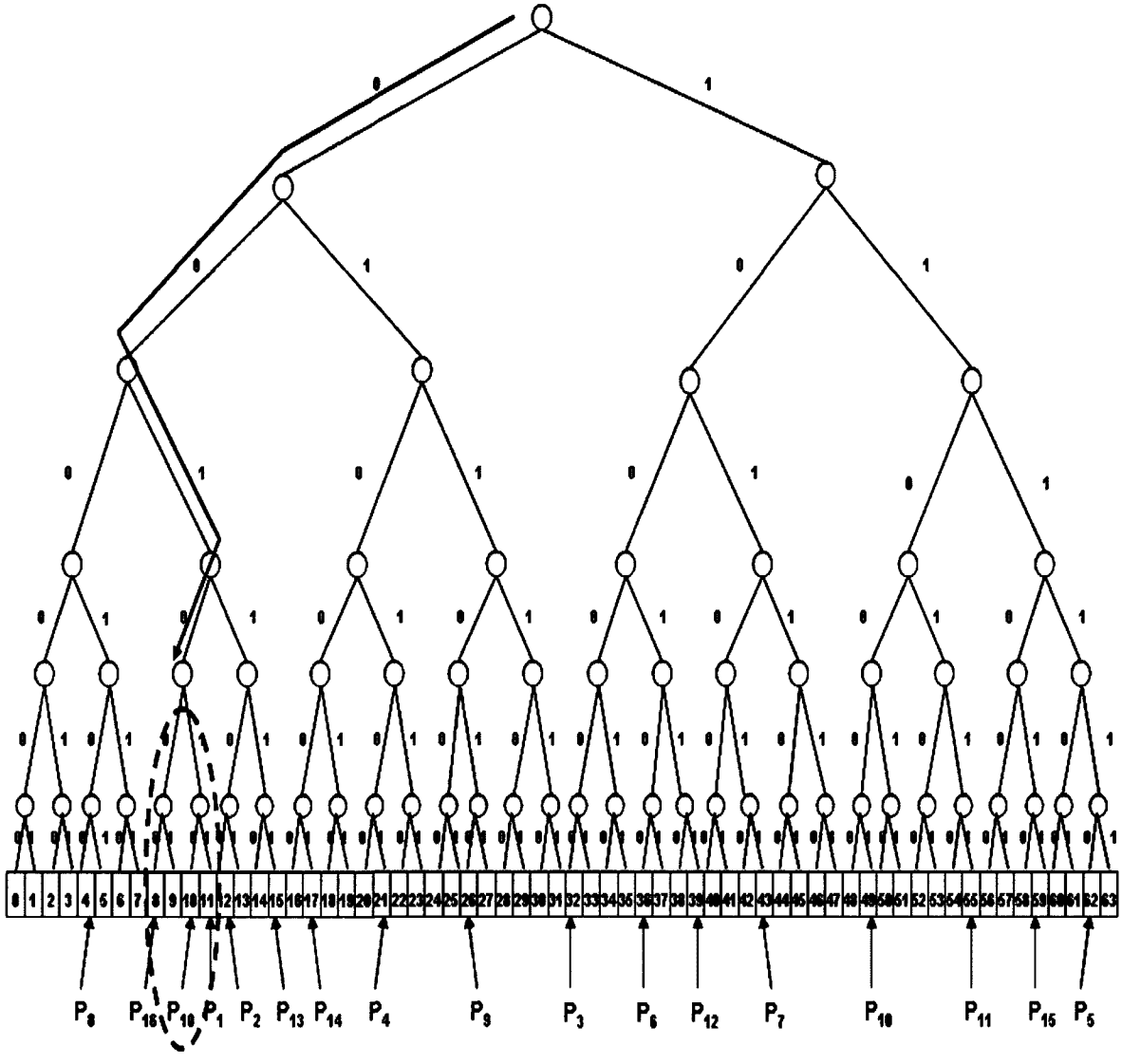


Figure 28.Range search example

Chapter 5

Network Dynamics

5.1 Introduction

In MMVE, each user is represented by an entity (avatar) whose state is controlled by user input and multi-user interaction is supported by matching user actions to entity updates (i.e. motion/sound generation) in the shared virtual environment. User movements in MMVEs need to be effectively supported, especially in applications like distributed training simulations, collaborative design, virtual meetings and multiplayer games.

When nodes move in the MMVE, they change their location in the virtual environment. Therefore, their NodeID which has been derived through mapping their 3-dimanisonal location to the Z-order curve changes too. The change in a node location might cause changes in its own routing table and the routing table of nodes who had this node as their neighbour. The update of routing tables when the network nodes frequently change their location is a challenging process, and it imposes a large overhead on the network.

Routing table updates can be performed in two ways, i.e. global way or selective way. While global update for the routing tables is much simpler, selective update is much more efficient when users move very often, therefore requiring frequent updates in the routing tables.

In the selective update approach, when a node moves, it will try to send selective routing table updates to nodes whose routing table entries related to that node need to be changed. It also gets the help from its neighbours or ex-neighbours to identify new neighbours, so it can update its own routing table while informing the old neighbours about the change.

In this chapter we describe how users' movements are supported by the proposed architecture that is based on users' location mapped by Z-order space filling curve and stored

in KD-tree data structure. As mentioned before, each row of the routing table is assigned a level. Here we refer to the nodes at level K of the routing table as level K neighbours. It should be noted that neighbours at different levels need to be treated differently in the routing table update procedure. Based on this neighbour level definition, we will explain our approach to the routing table update process in three important situations, i.e. when nodes join network, move in the network, and leave the network.

5.2 Node Arrival

When a new node arrives, it needs to initialize its routing table, and inform other nodes of its presence. We assume that joining node knows initially one of the nodes in the network that acts as a bridge helping to connect the joining node to the right point in the network. Such a node can be located automatically, for example, by using "expanded ring" IP multicast, or it can be obtained by the system administrator through outside channels.

After it joins the network, the joining node receives its NodeID based on its network location. Next, it contacts its known points and sends a query with its joining coordinates (NodeID and IP address). The query will eventually reach a node with a position closest to the joining node's position. This node has most of its ID bits in common with the joining node, so their routing tables should have lots of common entries. When it receives this joining query, the closest node to the joining node will then respond by sending its routing table information to the joining node. The joining node then organizes its own routing table, and connects to each neighbour in the routing table. The neighbours also update their routing tables to account for the joining node.

Next, we illustrate this process with an example. Let us assume that we have a network of nodes as illustrated in Figure 29, and that a new node with NodeID 45 joins the network. As described above, this new node sends a joining query and receives a response from the node that has a NodeID (i.e. position) closest to the joining node (in the scenario illustrated bellow, it is node with NodeID 43). The joining node organizes its routing table based on the routing table information it received from the closest node (shown in Table 3 and Table 4).

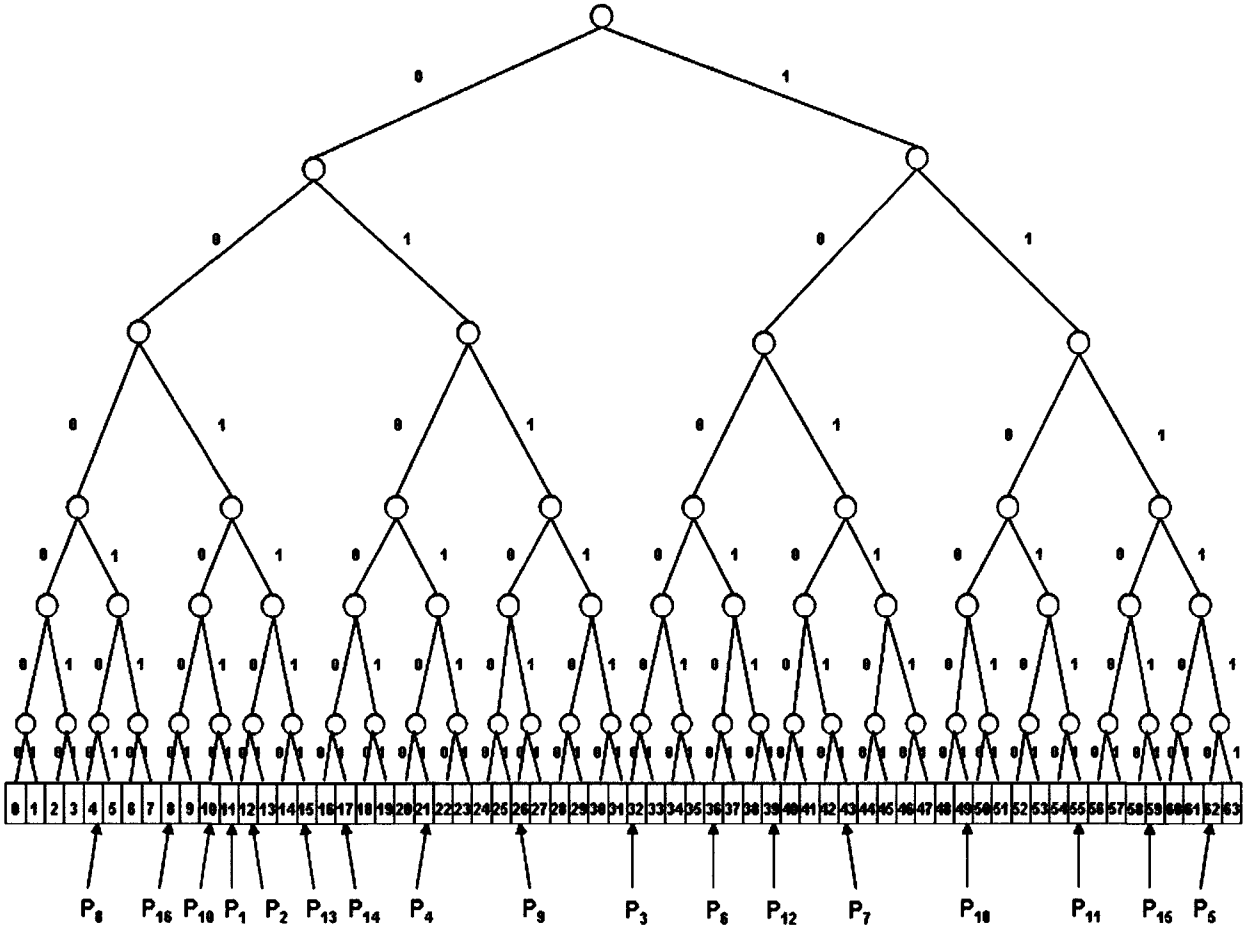


Figure 29. Network in KD-tree format for a node arrival example

Table 3. Routing table received from the closest node (NodeID 43)

	0	1	2	3
0	4	8	10	11
1	49	55	59	62
2	32	36	39	-
3	-	-	-	-
4	-	-	-	-
5	-	-	-	-

Table 4. Joining node's routing table (NodeID 45)

	0	1	2	3
0	4	8	10	11
1	49	55	59	62
2	32	36	39	-
3	43	-	-	-
4	-	-	-	-
5	-	-	-	-

As indicated previously, other nodes also update their routing tables to account for the joining node. For example, node with NodeID 43 adds joining node's information to its routing table (Table 3), and adds it to the appropriate level, i.e. since their IDs have 3 bits in common; it adds it to the level 3 in its routing table.

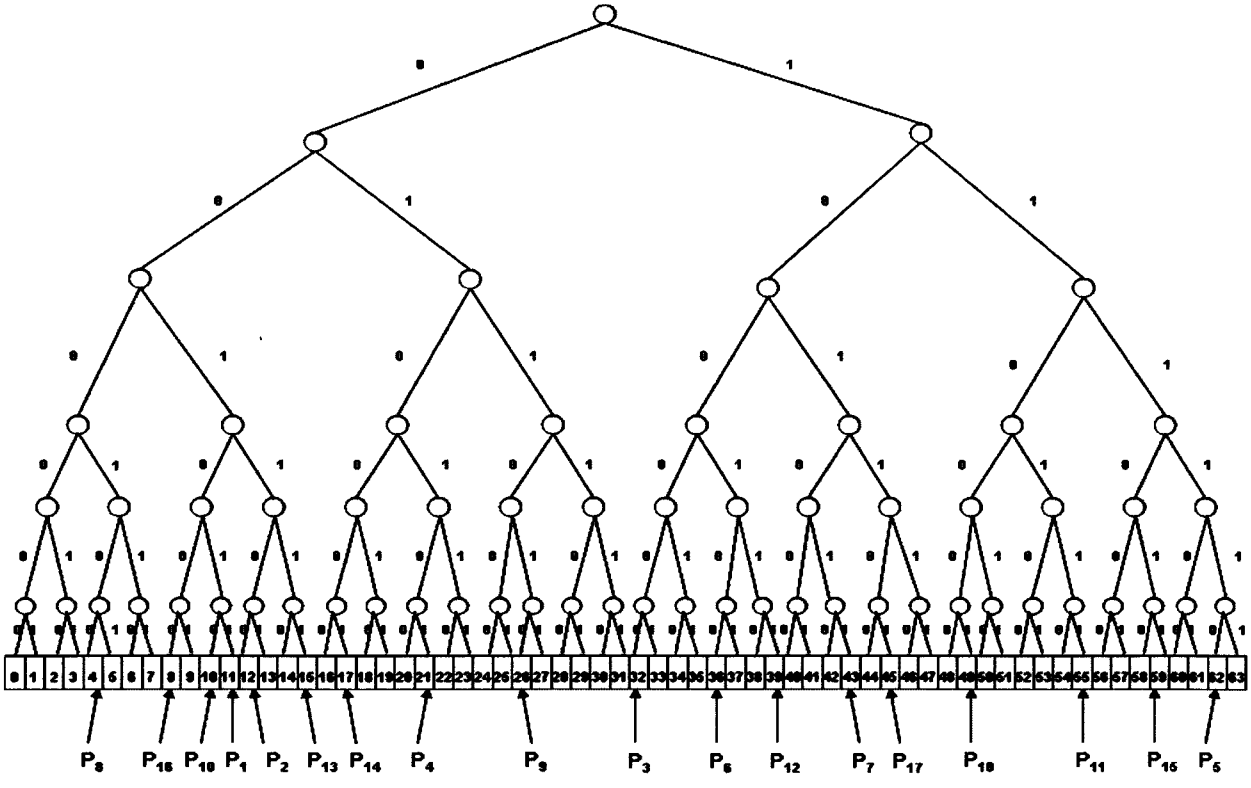


Figure 30. KD-tree after the join procedure

The arrival of new node has an impact on routing tables of several nodes in this example, specifically, nodes with NodeIDs 49, 55, 59 and 52 (since they are close to the newly joined node). At the end of the join procedure, the updated KD-tree data structure is illustrated in Figure 30.

5.3 Node Movement

As stated previously, node movement might cause changes in its own routing table and the routing table of nodes who had this node as their neighbour. The update of routing tables when the network nodes frequently change their location is a challenging process, and it might impose a large overhead on the network.

When a node moves, it receives a new NodeID based on its new position in the network. The position updates (i.e. new NodeID) are sent to all those nodes that have this node in their routing tables. The nodes that have the moving node in their routing table will check to see if they have to update their routing table or if the routing table still works as it is.

The neighbours of the moving node will perform different procedures based on their level of neighbourhood. Assuming that we have b bits for the NodeIDs, neighbours are assigned levels 0 to b – 1, where b – 1 level contains the closest neighbours (i.e. these nodes NodeIDs have b – 1 bits in common).

If the update recipient is a b – 1 level neighbour, it checks to see whether its NodeID still has b – 1 bits in common with NodeID of the moving node. If so, it will keep the moving node in its routing table; otherwise it will try to find a replacement for it.

If the update recipient is a b – 2 level neighbour, it will check if its NodeID still has b – 2 bits in common with NodeID of the moving node. If so, no updates to the routing table are necessary. On the other hand, if not, two scenarios are possible:

1. The update recipient node might have b – 1 bits in common with the moving node NodeID, so it will then inform the moving node of a set of new possible b – 1 level neighbours from its own routing table. Moreover, in this case, moving node can use information from this routing table to completely update its own routing table, since other level neighbours (0 to b – 2) are common, since their NodeIDs have b – 1 bits in common. The moving node will then disconnect from its old neighbours that are no longer eligible and connect to the new neighbours accordingly.

2. The update recipient NodeID has less than b – 2 bits in common with the moving node NodeID. It removes the moving node from that level of the routing table, and looks for a replacement (if necessary). It also checks to see what level of neighbourhood it has with the moving node based on new information (i.e. moving node new NodeID), let us label it with *l*, and it adds it to the appropriate level of its routing table (level *l*), if necessary, i.e. if there is an empty entry in that level. Also, it shares the routing table information with the moving node for levels 0 to *l*, since moving node should have those levels in the routing table populated with the nodes that have NodeID prefix exactly as it is in the update recipient node's routing table.

In general, the same procedure applies for every level of neighbourhood. The update recipient checks if the level of neighbourhood has changed, and if so, it updates its routing table accordingly. At the same time, based on this new neighbourhood level, it provides information to the moving node, so the moving node can adjust its routing table accordingly (as described above). Therefore, the moving node creates its new routing table based on information in its old routing table together with the information it receives from all the other nodes that were affected by the movement.

Let us look at an example of node movement. We assume that we have a network of nodes as presented in Figure 30 (in previous section). We take a look at node with NodeID 55 and its routing table (Table 5) and see what happens when this node changes its position as it is presented in Figure 31. We show how it affects its routing table, and routing table for some of the nodes in its old and new neighbourhood (for simplicity purposes we just show an

impact on routing table for one of the nodes in old neighbourhood and one of the nodes in new neighbourhood).

Table 5. Routing table for node (NodeID 55) prior to movement

	0	1	2	3
0	4	8	10	11
1	32	36	39	43
2	59	62	-	-
3	49	-	-	-
4	-	-	-	-
5	-	-	-	-

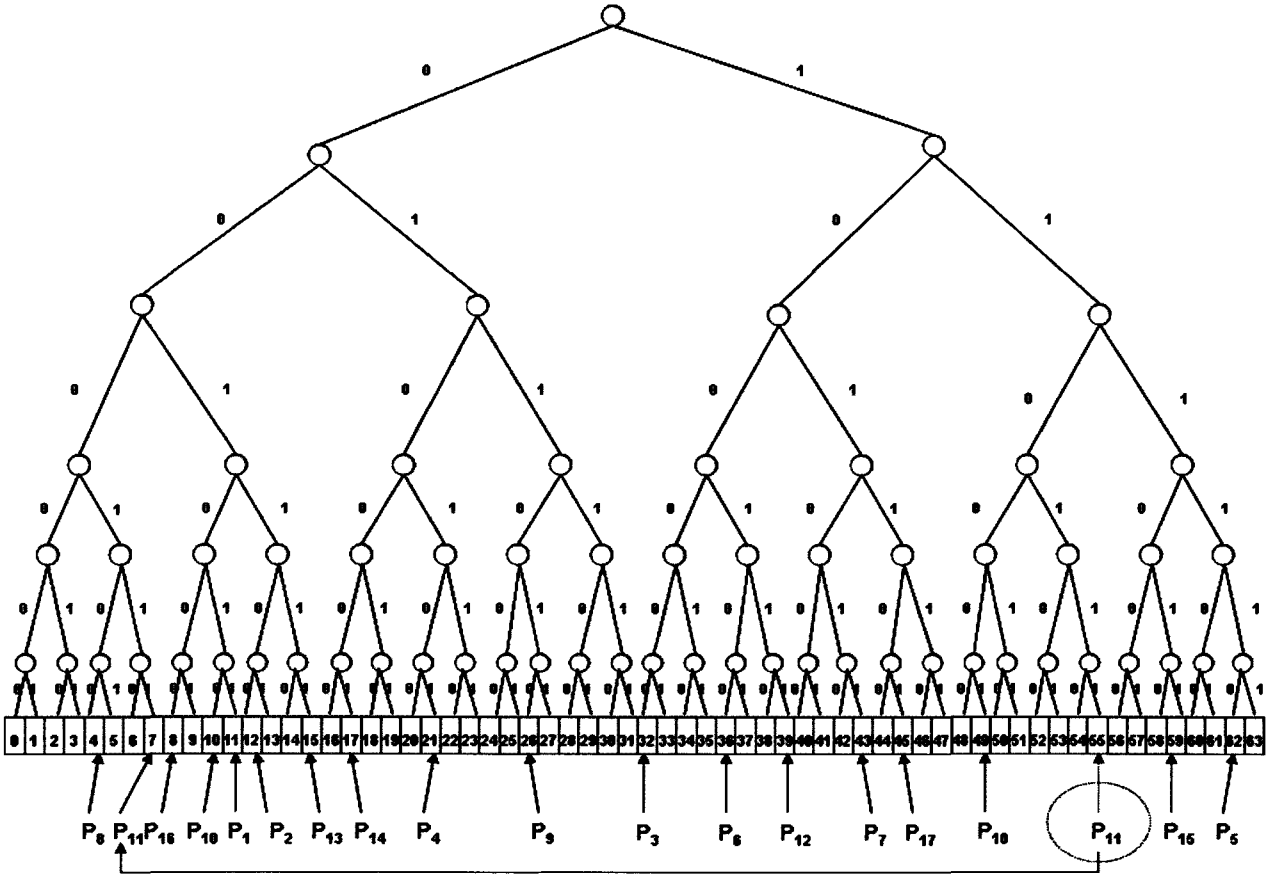


Figure 31.Example of node movement

As it is presented in Figure 31, the node receives a new NodeID based on its new position in the network. In the first step, the moving node checks for the nodes in its current routing table that have NodeID close to its new NodeID. It identifies that nodes located in row 0 of its routing table are good candidates for providing information for its new routing table. Specifically, it identifies node with NodeID 4 to have the most common bits with its new NodeID, and queries it for the routing information. It receives routing table information from the node with NodeID 4 (as presented in Table 6), and uses it to organize its new routing table.

Table 6.Routing table for node with NodeID 4 (prior to movement)

	0	**1**	**2**	**3**
0	32	36	39	43
1	17	21	26	-
2	8	10	11	12
3	-	-	-	-
4	-	-	-	-
5	-	-	-	-

Table 7.Routing table for node with NodeID 45 after the movement

	0	**1**	**2**	**3**
0	4	8	10	11
1	49	59	62	-
2	32	36	39	-
3	43	-	-	-
4	-	-	-	-
5	-	-	-	-

Also, the moving node checks the information it receives from the other nodes, i.e. the nodes that had to readjust their own routing tables as a result of this movement, since these nodes (as described before) send information on nodes that are at the same neighbourhood level in their routing tables as the moving node. As it can be seen in the example presented in Table 7 for node with NodeID 45, it removes moving node from level 1 in its routing table (see Table 4 to see the routing table before the movement), and it sends information on nodes in its level 0 (since moving node's NodeID has the most common bits with those nodes). It should be noted that node with NodeID 45 does not add moving node to its routing table at level 0, since it already has four entries in the table at that specific level. However, in the case that one of those entries was empty, it would add the moving node to its routing table (as described previously).

Therefore, the moving node uses information from its old routing table together with the information received from the closest identified node (i.e. node with NodeID 4) and all the other nodes that were affected by the movement to create its new routing table (presented in Table 8).

Table 8.New routing table for the moving node (NodeID 7)

	0	**1**	**2**	**3**
0	43	49	59	62
1	17	21	26	-
2	8	10	11	12
3	-	-	-	-
4	4	-	-	-
5	-	-	-	-

It should be noted that the node with NodeID 4 adds moving node to its routing table at level 4. Also, all the other nodes that are affected by the movement adjust their routing tables accordingly.

5.4 Node Departure

Node departure from the network can happen at any time. In the case of network departure, leaving node simply disconnects from the network. As topology discovery messages are regularly exchanged among the nodes, the neighbour nodes will discover the absence of a node as soon as it does not respond to topology discovery messages. The neighbour nodes would then try to find a replacement, if necessary, since they usually have more than one entry at each level of their routing table.

Chapter 6

Performance Evaluation

6.1 Introduction

MMVE needs to be highly responsive and to progress seamlessly, and update messages distribution among entities in the MMVE plays a major role in both of those areas. In this chapter we present experimental results for our P2P architecture and VERA routing algorithm that uses sophisticated addressing mechanism which minimizes network overhead and improves network latency and scalability.

Preliminary experimental results obtained from the initial development of the whole architecture are promising and show considerable reduction of routing hops. Simulations have been done in P2P networks where nodes are uniformly distributed in the network space. In our simulation we partitioned the network space using 9^{th} order 3-dimensional Z-order curve for multidimensional indexing.

6.2 Simulation Configuration

In order to do preliminary evaluation of latency and scalability for our architecture, and to obtain preliminary experimental results, we have simulated several components of our architecture. Specifically, we have implemented node IDs assignment based on nodes' 3D coordinates in the virtual environment (using 9^{th} order 3-dimensional Z-order curve for multidimensional indexing), then routing table construction for each node as described in section 4.3, and finally VERA, a novel virtual environment routing algorithm.

All experiments, i.e. simulations and testing were performed on 1.73 GHz Intel Pentium M processor and 512 MB of RAM, running Microsoft Windows XP. Simulation was implemented in Java (version 6) and executed using the Eclipse (version 3.2).

The preliminary experimental results and evaluation of our architecture (based on Z-order curve and KD-tree) and VERA routing algorithm are presented in the next section.

6.3 Analysis of Preliminary Results

MMVEs often host thousands of participants at any time, so it is critically important to have a highly scalable and flexible network with low network latency (i.e. low update delay). As stated previously, update messages distribution (i.e. routing algorithm) among entities in the MMVE plays a major role in overlay network scalability and flexibility. Therefore, we evaluate our architecture and our routing algorithm (VERA) by analyzing average number of routing hops as a function of size of the P2P network, and by measuring average delay for messages distribution among the nodes in the network.

The preliminary simulation results presented show the average number of routing hops as a function of size of the P2P network. We varied the number of nodes from 1,000 to 10,000, where nodes were uniformly distributed in the network space. In each trail we took the average of 1,000 runs (even though there was no much variation between simulation runs), and the average hop count for transferred messages has been recorded and results are presented in Figure 32. Also, we have compared the average number of routing hops with the log N value (where N is the number of network nodes).

The results presented here are promising. More specifically, for the specific setup presented in this section, in a MMVE network with 10,000 uniformly distributed nodes it takes, on average, 3.95 routing hops to send a message from randomly picked node A to randomly picked node B. Also, the results show that the average number of routing hops scale with the size of the network (i.e. log 10,000 = 4, where average number of hops for 10,000 nodes is 3.95).

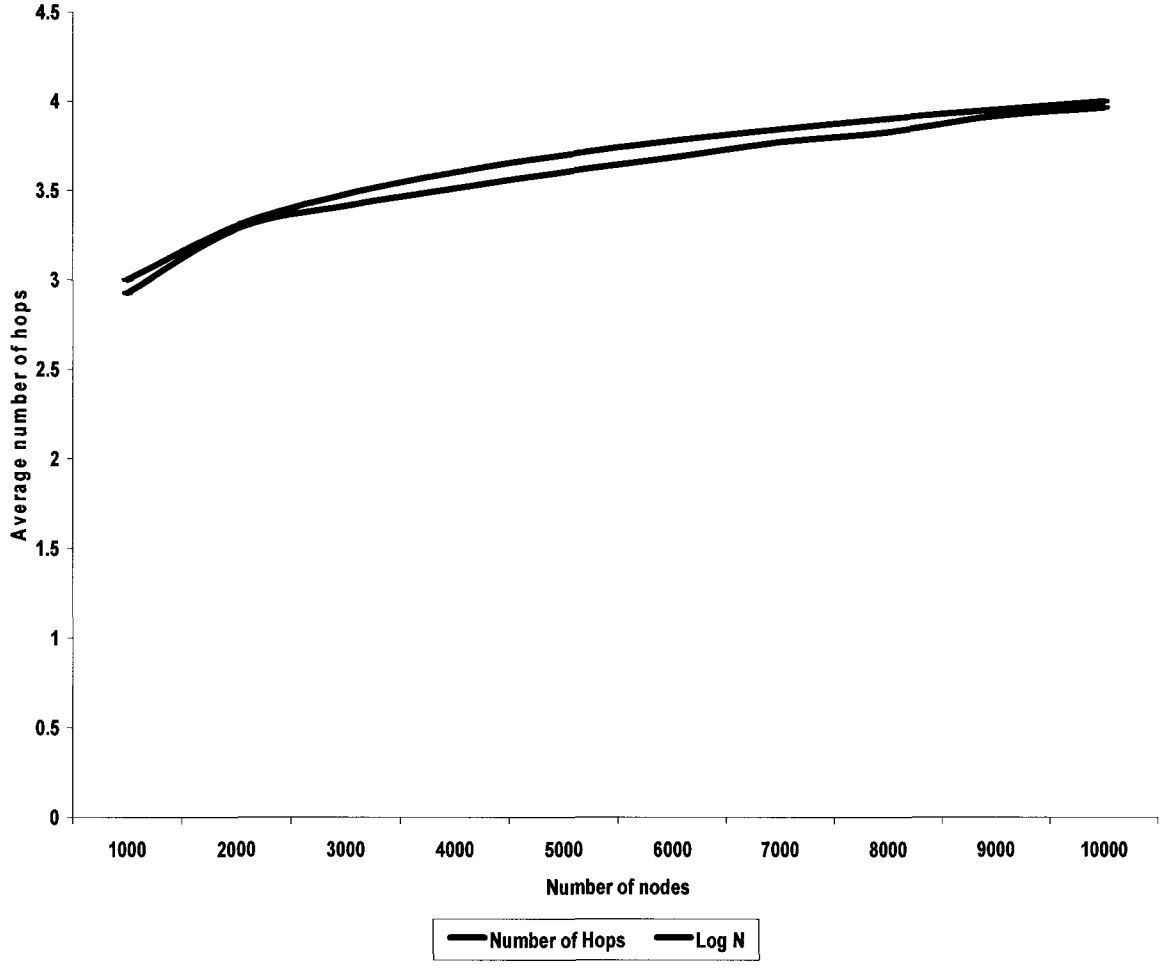


Figure 32.Average number of routing hops versus number of network nodes

Our architecture has several advantages over all other P2P architectures that do not use node's 3D location to do efficient routing (for example all peer-to-peer systems that support a distributed hash table (DHT) functionality, like Plaxton, Tapestry, Pastry, etc.).If we do a comparison of our architecture with one of the P2P systems that support DHT functionality, like Pastry, we can see several similarities, but also couple of advantages too.

Our routing performance preliminary results are similar to results for Pastry (presented in [20]). In [20] routing performance is also measured as the number of routing hops as a function of the size of the Pastry network (detailed information about the experiment setup is described in [20]), i.e. it shows the average number of routing hops taken as a function of the

network size. The results show that the number of routing hops for Pastry also scale with the size of the network, and it is close to O(log N).

However, our approach has several advantages over Pastry (and all the other P2P systems that support distributed hash table (DHT) functionality). More specifically, our architecture uses object's 3D virtual location to assign unique node identifiers (NodeIDs) which are used for efficient routing. On the other hand, in Pastry NodeIDs are generated such that the resulting set of NodeIDs is uniformly distributed in the 128-bit NodeID space, where these NodeIDs are generated by computing a cryptographic hash of the node's public key or its IP address. As a consequence of this type of random NodeID assignment, there is a high probability that nodes in Pastry that have adjacent NodeIDs are diverse in geography, ownership, jurisdiction, network attachment, etc. Therefore, two Pastry nodes that have NodeIDs close to each other can be physically far away from each other in the network which can affect network latency.

Also, in MMVEs very often communication must only occur within subsets of distributed entities, i.e. certain area of interest, that are either neighbourhood based or region based. Consequently, for a P2P, it is quite important to be able to identify theses area of interest, i.e. to perform range searches. Therefore, since our architecture uses node's network location, it is able to perform range searches (based on nodes location), whereas Pastry and all the other P2P systems that support distributed hash table (DHT) functionality cannot perform them.

Next we measured average delay for messages distribution among the nodes in the network. As before, we varied the number of nodes from 1,000 to 10,000 (nodes were uniformly distributed in the network space). In each trail we took the average delay of 1,000 runs for message transfer between randomly picked nodes A and B. The results are presented in Figure 33.

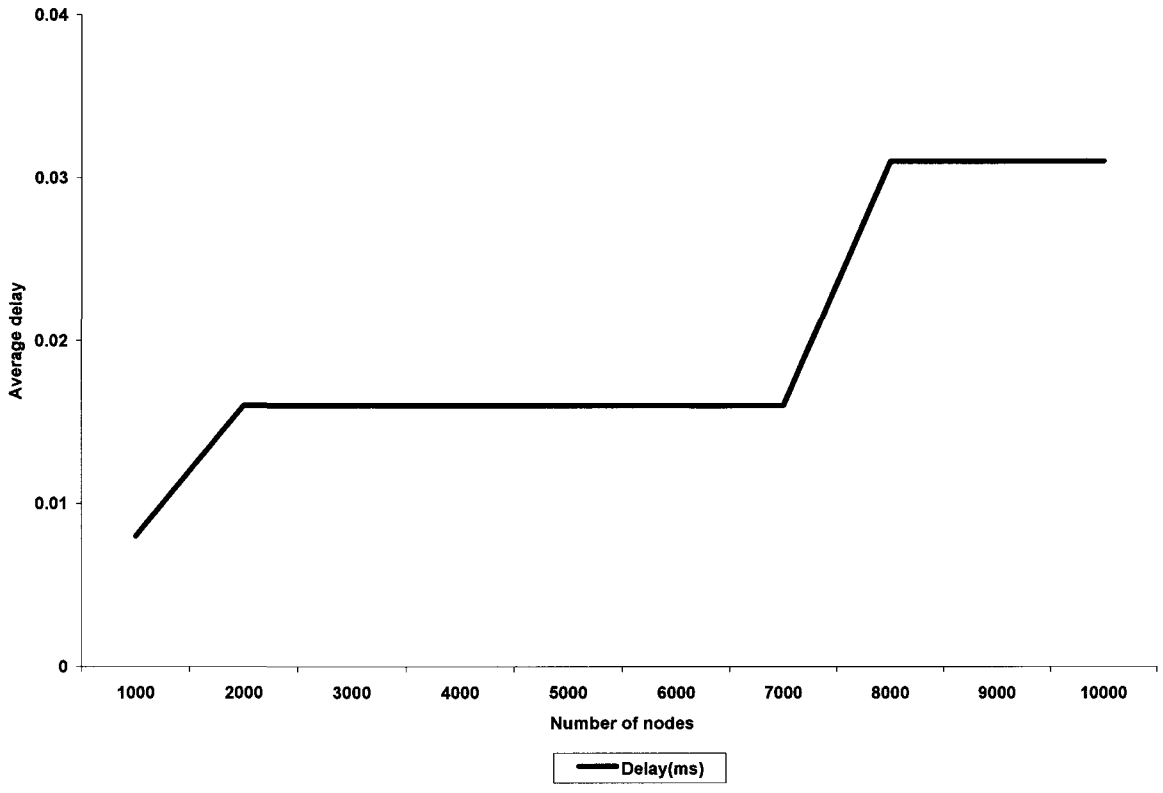


Figure 33.Average delay versus number of network nodes

As it can be seen in Figure 33, for the specific setup presented in this section, in a MMVE network with 10,000 uniformly distributed nodes, the average delay to send a message from randomly picked node A to randomly picked node B is 0.031 ms. Based on this simulation of network delay, it seems that every time certain threshold of network nodes is reached, the delay increases slightly and remains constant until another threshold is reached. However, further evaluation of the delay is required, since the delay presented here is only a delay for our simulation.

As a part of evaluation process of our architecture and our routing algorithm, we analyzed a number of node inspections required during a routing process from randomly picked node A to randomly picked node B. Specifically, we graphed the average number of nodes inspected against the number of nodes in the network, where node was counted as being inspected if the distance between its NodeID and the destination NodeID was computed. In our case, i.e.

during the execution of VERA algorithm, the distance is being computed when entries in a specific row in the routing table during the routing process are being checked to calculate which of the entries in that row of the routing table has the NodeID value closest to the destination NodeID.

The setup for this part of evaluation was exactly the same as for the previous two evaluations (i.e. average number of hops and average delay), and as before in each trail we took the average of 1,000 runs, and the average number of inspections has been recorded. The results are presented in Figure 34.

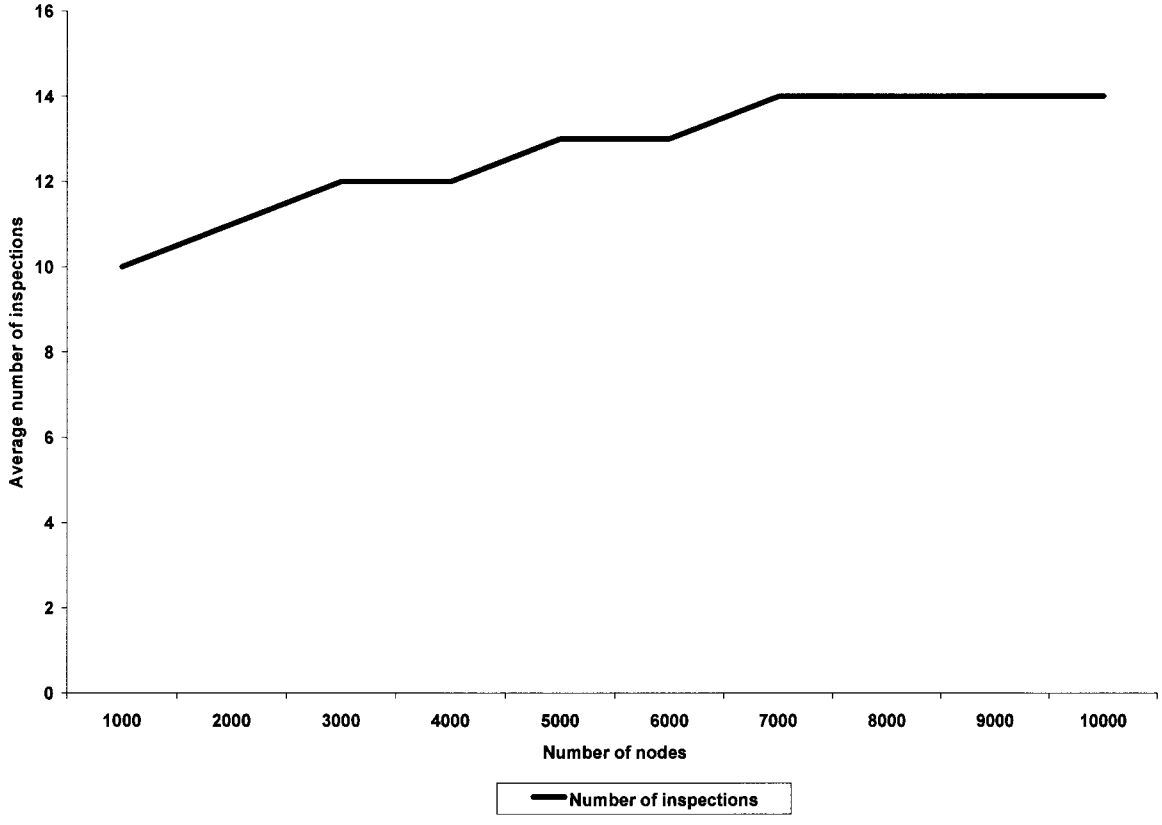


Figure 34.Average number of inspections versus number of nodes

As it can be seen from results presented in Figure 34, number of inspections increases slowly as the number of nodes in the network increases. Specifically, for a MMVE network with 10,000 uniformly distributed nodes, the average number of inspections is 14. In general, for our architecture, number of inspections is dependent from number of routing hops, since for

each routing hop minimum of 1 and maximum of 4 node inspections is possible (since in our architecture, each row of the routing table has at most 4 entries). For example, if we look at the case with 10,000 uniformly distributed nodes, we have 3.95 average number of routing hops and 14 node inspections, which means that (for this specific example) there are approximately 3.5 node inspections per hop.

The number of node inspections is usually used to evaluate various searching algorithms for KD-trees or any other data structures that are commonly used to perform different searching algorithms. In general, for our architecture, to find a specific node in a network of N nodes it is clear that at most O(h) nodes need to be inspected, where h is a height of the KD-tree used to represent our network space, since each routing table has h rows .

Chapter 7

Conclusions and Future Work

In recent years, massively multi-user virtual environments (MMVE) have been a major research topic for the virtual reality community. MMVE applications incorporate computer graphics and sound simulation, together with the achievements in the networking field that further motivated multi-user interaction over the internet, to simulate the experience of real-time interaction between multiple users in a shared three-dimensional (3D) virtual world. Multi-user virtual environment technology can be applied in many areas, for example, distributed training, simulation, education, home shopping, virtual meetings, and massive multi player online games (MMOGs).

In order for the MMVE to be highly responsive and to progress seamlessly, entities interacting in the shared virtual environment must update each other frequently. Every time there is a change related to entity's state or a modification to the shared virtual environment or impact on the other entities, an update message needs to be sent to inform other entities in the shared virtual environment. With a significant increase of virtual environment users and entities, there is a related increase in the number of update messages that are being exchanged among the users and entities. As a result, there is a major increase in the importance of effective update messages distribution on the overlay framework of the virtual environment, while achieving an acceptable level of latency is also a concern in the update messages exchange process.

Therefore, as stated previously, research objective of this thesis can be summarized as follows:

- Efficient communication (i.e. update messages distribution) among entities in MMVEs.

- Due to the fact that entities' area of interest usually lies within certain visibility range, the routing algorithm should be based on actual entities' positions in the virtual environment.

- Network latency reduction

- Improved network scalability

In this thesis we have presented a new architecture for MMVEs that reduces the real latency for different lookup operations and provides efficient communication among network peers. It is based on the location of network nodes (i.e. 3-dimensional coordinates), and it uses Z-order space filling curve together with KD-trees to efficiently construct routing tables. Z-order space filling curve is used to perform multidimensional indexing, i.e. the method of mapping the points of the multidimensional space to a one-dimensional index. Specifically, an object's virtual environment 3D coordinates are mapped to the nearest vertex of the Z-order curve, and the index number of the vertex is then used as the object's key within the network overlay. Since KD-trees have been shown well suited for range searching and they also group the data points together alternating the discriminating keys, it is obvious that KD-trees are a suitable data structure for presenting one-dimensional order of multidimensional data points. Therefore, this two step process is used to construct routing tables that are used by our routing algorithm (VERA) which uses each node's routing table to perform an efficient update messages distribution. We have also developed an approach for efficiently handling networks dynamics, specifically, join, movement, and departure process for this newly proposed P2P architecture.

Our architecture has several advantages over all other P2P architectures that do not use node's 3D location to do efficient routing (for example all peer-to-peer systems that support a distributed hash table (DHT) functionality, like Plaxton, Tapestry, Pastry, etc.). Specifically, in MMVEs very often communication must occur only within subsets of distributed entities, i.e. certain area of interest, that are either neighbourhood based on region based. Therefore, we use Z-order curve, since it has good locality preserving behaviour and

groups close by multidimensional points together (a feature desirable when searching for compact ranges), together with KD-trees, as they also group data points together and perform range searches efficiently, to perform efficient routing. Therefore our architecture, since it uses node's network location, is able to perform range searches (based on nodes location), whereas Pastry and all the other P2P systems that support distributed hash table (DHT) functionality cannot perform them. Therefore, in this thesis we have also presented a range search process that is based on the proposed P2P architecture (i.e. Z-order curve and KD-tree). Since our architecture is based on object's location in the virtual environment, i.e. 3D coordinates, our search process can be used for both neighbourhood based search or region based search.

Preliminary experimental results have demonstrated that this approach can considerably decrease the routing cost among nodes in MMVE network, and it can be very valuable for many multiuser virtual environment applications.

In our future work, we will continue to further evaluate and analyze the proposed architecture. Specifically, we need to further explore and analyze network dynamics, i.e. join, movement, and departure process for this newly proposed P2P architecture. Also, it would be beneficial to continue further analysis and comparison of VERA with other routing algorithms.

Also, there are many different applications where our architecture can be applied. Specifically, our architecture can be applied in any P2P decentralized networks, so it can be very valuable for many decentralized distributed applications, like underwater sensor networks, geographical observation systems based on P2P agents, surveillance systems for high rises, mobile systems, etc. All these P2P networks have some sort of location based communication and services. For example, in mobile systems, the users need to access information about wireless cells across different management domains and about different access technologies. The P2P overlay aggregates meta-data of heterogeneous cellular networks according to their geographic coverage, so mobile users can start a location-based range query to find other accessible cells within their movement range. Therefore, future

work should identify all the special requirements for different P2P applications (like the ones mentioned above), and evaluate how our architecture and routing algorithm can be modified to satisfy each of these specific requirements and improve their performance.

References

[1] S. Zonjic, B. Hariri, S. Shirmohammadi, "Multidimensional query based routing for virtual environments," VECIMS 2009, May 11-13, 2009, Hong Kong, China.

[2] T. A. Funkhouser, "Network topologies for scalable multi-user virtual environments," IEEE VRAIS '96, April, 1996, San Jose, CA, USA, pp. 222-228.

[3] B. Knutsson, H. Lu, W. Xu, and B. Hopkins "Peer-to-peer support for massively multiplayer games," in Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM 2004, March 2004, pp. 96-107.

[4] T. Hampel, T. Bopp, and R. Hinn, "A peer-to-peer architecture for massive multiplayer online games," in Proceedings of the 5th ACM SIGCOMM workshop on Network and system support for games, Netgames 2006, Singapore, October 2006, pp. 48-51.

[5] C. Diot and L. Gautier, "A distributed architecture for multiplayer interactive applications on the Internet," IEEE Network, vol. 13, no. 4, 1999, pp. 6-15

[6] S. Douglas, E. Tanin, A. Harwood, and S. Karunasekera, "Enabling massively multi-player online gaming applications on a P2P architecture," in Proceedings of the International Conference on Information and Automation, ICIAD 2005, December 2005, Colombo, Sri Lanka, pp. 7-12

[7] T. Limura, H. Hazeyama, and Y. Kadobayashi, "Zoned federation of game servers: a peer-to-peer approach to scalable multi-player online games," NetGames-04 (SIGCOMM Conference 2004) Workshops, August 30th – September 3rd, 2004, Portland, Oregon, USA.

[8] A. Yu and S.T. Vuong, "MOPAR: A mobile peer-to-peer overlay architecture for interest management of massively multiplayer online games," in Proceedings of the Network and Operating System Support for Digital Audio and Video Conference (NOSSDAV 2005), June 13-14, 2005, Stevenson, Washington, USA, pp. 99-104.

[9] A. Bharambe, J. Pang, and S. Seshan, "Colyseus: A distributed architecture for online multiplayer games," in the Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI 2006), May 08-10, 2006, San Jose, CA, USA, pp. 155-168.

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

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

[12] C. G. Plaxton, R. Rajaraman , and A. W. Richa, "Accessing nearby copies of replicated objects in a distributed environment," in Proceedings of ACM SPAA, Newport, Rhode Island, June 1997, pp. 311-320.

[13] S. Ratnasamy, S. Shenker, and I. Stoica, "Routing algorithms for DHTs: Some open questions," in Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS 2002), March 7-8, 2002, MIT Faculty Club, Cambridge, MA, USA, pp. 45-52.

[14] M. Harren, J. M. Hellerstein, R. Huebsch, B. T. Loo, S. Shenker, and I. Stoica, "Complex queries in DHT-based peer-to-peer networks," in Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS 2002), March 7-8, 2002, MIT Faculty Club, Cambridge, MA, USA, pp. 242-250.

[15] M. J. Freedman and R. Vingralek, "Efficient peer-to-peer lookup based on a distributed trie," in Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS 2002), March 7-8, 2002, MIT Faculty Club, Cambridge, MA, USA, pp. 66-75.

[16] C. Zheng, G. Shen, S. Li, and S. Shenker, "Distributed segment tree: Support of range query and cover query over DHT," the 5th International Workshop on Peer-to-Peer Systems (IPTPS 2006), February 27-28, 2006, Santa Barbara, CA, USA.

[17] K. Park, S. Pack, and Y. Choi, "Proximity based peer-to-peer overlay networks (P3ON) with load distribution," in Proceedings of the International Conference on Information Networking (ICOIN 2007), January 23-25, 2007, Estoril, Portugal, pp. 234-243.

[18] B. Y. Zhao, L. Huang, J. Stribling, S.C. Rhea, A.D. Joseph, and J.D. Kubiatowitcz, "Tapestry: A resilient global-scale overlay for service deployment," IEEE Journal on Selected Areas in Communications, vol. 22, no. 1, pp. 41-53, January 2004.

[19] B. Y. Zhao, J.D. Kubiatowitcz , and A.D. Joseph, " Tapestry: An infrastructure for fault-tolerant wide-area location and routing," Technical Report UCB/CSD-01-1141, University of California at Berkeley, Computer Science Department, 2001.

[20] Rowstron, A., Druschel, P., "Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems." In Proc. of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001), Heidelberg, Germany, November 2001.

[21] 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 SIGCOMM, San Diego, CA, Aug 2001, pp.149-160.

[22] D. Karger, E. Lehman, E. Leighton, F. Levine, M. D. Lewin, and R. Panigrahy, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web," in Proceedings of the 29th Annual ACM Symposium on Theory of Computing, El paso, TX, May 1997, pp. 654-663.

[23] FIPS 180-1. Secure Hash Standard. U.S. Department of Commerce/NIST, National Technical Information Service, Springfield, VA, April 1995.

[24] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, " A scalable content-addressable network," in Proceedings of SIGCOMM, San Diego, CA, Aug 2001, pp. 161-172.

[25] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system basd on the XOR metric," in Preceedings of the IPTPS, Cambridge, MA, USA, February 2002, pp. 53-65.

[26] D. Malkhi, M. Naor, and D. Ratajczak, "Viceroy: a scalable and dynamic emulation of the butterfly ," in Proceedings of the ACM PODC 2002, Monterey, CA, USA, July 2002, pp. 183-192.

[27] H. J. Siegel, "Interconnection networks for simd machines," Computer, vol. 12, no.6, pp. 57-65, 1979.

[28] J. Kleinberg, "The small-world phenomenon: an algorithm perspective," in Proceedings of the 32nd annual ACM symposium on theory of computing, 2000, pp.163-170.

[29] L. Barriere, P. Fraigniaud, E. Kranakis, and D. Krizanc, "Efficient routing in networks with long range contacts," in Proceedings of the 15th International Conference on Distributed Computing, vol. 2180, 2001, pp. 270-284.

[30] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, S. Lim, "A survey and comparison of peer-to-peer overaly network schemes," IEEE Communications Survey and Tutorial, Volume 7, Issue 2, Second Quarter 2005, pp. 72-93.

[31] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron, "Topology-aware routing in structured peer-to-peer overlay networks," Microsoft Research, Technical Report, MSR-TR-2002-82, Redmond, WA 98052, 2002.

[32] G. M. Morton, "A computer oriented geodetic data base; and a new technique in file sequencing," Technical Report, Ottawa, Canada, 1966, IBM Ltd.

[33] J. A. Orenstein and F. A. Manola, "PROBE Spatial data modeling and query processing in an image database application," IEEE transactions on software engineering, vol. 14, no. 5, May 1988, pp. 611-629.

[34] J.K. Lawder, P.J.H. King, "Using space-filling curves for multi-dimensional indexing." Lecture Notes in Computer Science, Volume 1832/2000, Springer Berlin/Heidelberg, 2000, pp. 20-35.

[35] H. V. Jagadish, "Linear clustering of objects with multiple attributes," in Proceedings of ACM SIGMOD, June 1990, pp. 332-342.

[36] H. Tropf, H. Herzog, "Multidimensional range search in dynamically balanced trees," Applied Informatics, Vieweg Verlag, Wiesbaden, Germany, February 1981, pp. 71-77.

[37] J. L. Bentley, "Multidimensional binary search trees in database applications." IEEE Transactions on Software Engineering, Vol. Se-5, No. 4, July 1979.

[38] J. L. Bentley, "Multidimensional binary search trees used for associative searching," Communications Association for Computing Machinery, Vol. 19, September 1975, pp. 509-517.

[39] J. J. Friedman, J. L. Bentley, and R.A. Finkel, "An algorithm for finding best matches in logarithmic expected time," ACM Trans. Mathematical Software, Vol. 3, September

1977, pp.209-226.

[40] T. H. Cormen, C.E. Leiserson, R. L. Rivest, "Introduction to Algorithms (2nd edition)", MIT Press and McGraw-Hill, Chapter 10, 2001.