Towards an Efficient Indexing and Searching Model for Service Discovery in a Decentralised Environment

Dejun Miao

A Thesis Submitted in Fulfilment of the

Requirements for the Degree of

Doctor of Philosophy

University of Derby

College of Engineering and Technology

2018

Contents

List of Figures

List of Tables

List of Algorithms

List of Equations

List of Definitions

List of Publications

Journal Papers

1. Dejun Miao, Rongyan Xu, Lu Liu, John Panneerselvam, Yan Wu and Wei Xu, An Efficient Indexing Model for Fog Layer of Industrial Internet of Things, IEEE Transactions on Industrial Informatics, 30 January 2018, DOI:10.1109/TII.2018.2799598 (*impact fact 6.764*)

2. Zhao Xu, Yan Wu, Dejun Miao and Lu Liu, A novel multilevel index model for distributed service repositories, Tsinghua Science and Technology, Year: 2017, Volume: 22, Issue: 3, DOI:10.23919/TST.2017.7914199, ISSN: 1007-0214, Pages: 273 – 281 *(impact fact 1.063)*

3. Yan Wu, Wei Xu, Lu Liu and Dejun Miao, Performance formula-based optimal deployments of multilevel indices for service retrieval, Concurrency and Computation Practice and Experience, DOI 10.1002/cpe.4265, *In-press (Impact fact 1.133)*

Conference Papers

4. Dejun Miao, Rongyan Xu, Lu Liu; Yan Wu, Paul Comerford and Zhao.Xu, A Novel Efficient Index Model and Modified Chord Protocol for Decentralised Service Repositories, 2017 3rd IEEE International Conference on Cybernetics (CYBCONF), Jun 21-23 2017, Exeter UK, IEEE Computing Society Press, DOI:10.1109/CYBConf.2017.7985748 ISBN: 978-1-5386-2202-5, Pages: 1-10

5. Rongyan Xu, Dejun Miao, Lu Liu and John Panneerselva, An Efficient and Shortest Route Plan for Yangzhou Based on Improved Floyd Algorithm, 2017 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), Jun 21-23 2017, Exeter UK, IEEE Computing Society Press, DOI: 10.1109/iThings-GreenCom-CPSCom-SmartData.2017.30**,**Pages:168 -177

Abstract

Given the growth and outreach of new information, communication, computing and electronic technologies in various dimensions, the amount of data has explosively increased in the recent years. Centralised systems suffer some limitations to dealing with this issue due to all data is stored in central data centres. Thus, decentralised systems are getting more attention and increasing in popularity. Moreover, efficient service discovery mechanisms have naturally become an essential component in both large-scale and small-scale decentralised systems and. This research study is aimed at modelling a novel efficient indexing and searching model for service discovery in decentralised environments comprising numerous repositories with massive stored services. The main contributions of this research study can be summarised in three components: a novel distributed multilevel indexing model, an optimised searching algorithm and a new simulation environment.

Indexing model has been widely used for efficient service discovery. For instance; the inverted index is one of the popular indexing models used for service retrieval in consistent repositories. However, redundancies are inevitable in the inverted index which is significantly time-consuming in the service discovery and retrieval process. This theeis proposes a novel distributed multilevel indexing model (DM-index), which offers an efficient solution for service discovery and retrieval in distributed service repositories comprising massive stored services. The architecture of the proposed indexing model encompasses four hierarchical levels to eliminate redundancy information in service repositories, to narrow the searching space and to reduce the number of traversed services whilst discovering services.

Distributed Hash Tables have been widely used to provide data lookup services with logarithmic message costs which only require maintenance of limited amounts of routing states. This thesis develops an optimised searching algorithm, named Double-layer No-redundancy Enhanced Bi-direction Chord (DNEB-Chord), to handle retrieval requests in distributed destination repositories efficiently. This DNEB-Chord algorithm achieves faster routing performances with the double-layer routing mechanism and optimal routing index.

The efficiency of the developed indexing and searching model is evaluated through theoretical analysis and experimental evaluation in a newly developed simulation environment, named Distributed Multilevel Bi-direction Simulator (DMBSim), which can be used as cost efficient tool for exploring various service configurations, user retrieval requirements and other parameter settings.

Both the theoretical validation and experimental evaluations demonstrate that the service discovery efficiency of the DM-index outperforms the sequential index and inverted index configurations. Furthermore, the experimental evaluation results demostrate that the DNEB-Chord algorithm performs better than the Chord in terms of reducing the incurred hop counts. Finally, simulation results demonstrate that the proposed indexing and searching model can achieve better service discovery performances in large-scale decentralised environments comprising numerous repositories with massive stored services.

Declaration

The study outlined in the dissertation was carried out in the College of Engineering and Technology at the University of Derby, under the supervision of Professor Lu Liu. This is to declare that the work carried out in this thesis was done by the author unless otherwise is indicated and no part of the thesis has been submitted in a thesis form to any other university or similar institution.

Dejun Miao

University of Derby

Derby, April 2018

Acknowledgement

The author would like to appreciate all those people who made this thesis possible and overwhelming experience.

First of all, the author is grateful to his PhD supervisor Prof Lu Liu for giving him the opportunity to study a PhD at the University of Derby, UK. This thesis would never have been possible without his help and support.

Secondly, the author expresses his special thanks to research colleagues and staff at the University of Derby, Prof Yong Xue, Dr Andrew Ramsey, Dr Xiaojun Zhai, Dr John Panneerselvam, Dr Bo Yuan and Dr James Hardy for their valuable suggestions and help.

Also, the author would express his gratitude to Dr Yan Wu, Jiangsu University, China; Dr Yankai Liu, Zhengzhou University, China; Dr Fang Tao, Dr Xinzheng Bai, Dr Paul Comerford, Miss Wei Xu and Mr Zhao Xu for their kindly help.

Finally, the author is indebted to his wife Mrs Rongyan Xu and daughter Miss Yujie Miao, to his parents Mr Lishan Miao and Mrs Guilan Qiao, to his father in law Mr Xian Xu and mother in law Mrs Minqun Lv during this study which supported him each step of the way. Their encouragement in many instances has been indispensable.

1 Introduction

This chapter introduces the research background, motivation and the aim and objectives of this thesis. The context of service discovery and the challenges involved in achieving an efficient solution for service discovering in decentralised environments without affecting the desired service quality are introduced. It further presents the research objectives and lists the major potential contributions of this study, including the development of an efficient indexing and searching model for service discovery in decentralised environments comprising a large number of service repositories with massive stored services. The organisation of this thesis is presented at the end of this chapter.

1.1 Research Background

In the recent decades, with the development of the Information and Communications Technology (ICT), the amount of data has grown exponentially. To allow faster and better integration of applications, software industry is going through a multitude of development perspectives [1]. Service-Oriented Architecture (SOA) has been widely implemented recently in various application domains, which also incurs ever-more-complex challenges. SOA is a logical way of designing a software system in order to provide services either to end-user applications or to assist other services distributed on a network via published and discoverable interfaces [2][3]. In SOA, all functions are defined as services and all services are autonomous, such that the services are usually defined self-contained modules [1][4].

As the most promising service-oriented computing (SOC) based technology, web services have proved to be the preferred implementation technology for

realising the SOA objectives of maximum service sharing, reuse and interoperability [5][6]. In web services, service requests are the messages formatted according to SOAP. Requested operations are implemented using one or more web service components which may be hosted within a web-services container [4][7][8].

With the development and deployment of Internet of Things (IoT) and social media networks, the size of service repositories has increased in the recent years [9]. More and more services are being generated by sensors, purchase transactions, mobile communication, search engines and so on. According to Gartner forecasts [10], more than 20.8 billion things will be connected by the year 2020 and 5.5 million new things are connected every day in last year. Beyond these predictions, McKinsey Global Institute [11] reported that the number of connected machines (units) has grown 300% over the last five years. According to the estimation and prediction of Cisco [12], the expected number of interconnected devices is to reach 50 billion by the year 2020. M. Hilbert and P. López [13] believed that 2.5 quintillion bytes of data are created every day.

Cloud Computing (CC), a widely employed approach that provides a computer-based environment where a range of computational services are available to be consumed by users online. It is considered as a promising computing paradigm, that can provide elastic resources to applications running on the end devices [14]–[18]. Nowadays, CC has been widely adopted as an efficient solution for those applications requiring rapid deployment of high_cost/low_use hardware and software resources. In centralised architectures, resources are hosted on the remote cloud datacentres and all the clients should send their requirements and receive responses over the network. Moreover, failures in remote cloud datacentres crash the entire network and none of the clients could access resources. Furthermore, centralised

architectures may easily create network bottlenecks when the system faces increasing number of resources wait processing. In practice, most of the remote cloud datacentres are geographically centralised and situated far from the proximity of users. But this service structure most often incurs additional process delays, network congestion, unacceptable round-trip latency and poor service quality etc; and the worse performance impacts are witnessed in the cases of real-time latency-sensitive applications.

To resolve those issues, a new concept named Fog Computing (FC) has emerged recently [19][20]. Cisco proposed a new computing paradigm, termed FC, that addesses the shortcomings of CC by transferring some of the core functionalities of cloud datacentres towards edge devices [21]. FC is a distributed computing paradigm that empowers the network devices with various degrees of computational and storage capability at different hierarchical levels [22]. It can act as a bridge between users and large-scale CC storage services. Through FC, it is possible to extend CC services to the edge devices of the network [23]. Compared to the cloud datacentres, FC has the potential to offer services with reduced services delays more than the CC service model [10], [23], [24].

With the growth of new technologies in various dimensions, decentralised networks have emerged as a promising computing paradigm. Peer-to-Peer (P2P) network is a typical distributed networking systems where peers (nodes) employ distributed resources to perform networking functionalities in a decentralised fashion, unlike the traditional centralised architectures which rely on a centralised servers [25]–[27]. P2P networks are usually large-scale networks comprising potentially millions of nodes, for instance, a trace of the Gnutella network carried out over eight days captured 1,239,487 peers back in the year 2001 [28][29]. In theory, nodes in P2P networks play equal roles and

not only can they act as network resources for a single client but also can serve as servers for others [25], which means that the resources can be shared between clients directly without a centralised server. However, it is no longer guaranteed that a requestor will be directly connected to a provider, which means that the required resources may be found by traversing the network via multiple hops. In practice, it is rare for a network to have a pure P2P architecture. Many such networks are hybrids: between pure P2P and client/server. Not all nodes in hybrid P2P architectures have equal role and responsibilities in the network like a pure P2P network [30]. Some nodes may take on additional responsibilities and become group managers or routers in spite of holding more information about the network and such nodes can route objects in fewer hops [30].

1.2 Research Motivation

The decentralised environments are considered as promising paradigms that can solve the contradiction between limited storage capacity and massive storage requirements faced by centralised systems. As discussed by Y. Wu et al. [31][32], service discovery is the process of finding one or more services whereas service composition is the means of finding and executing the services sequentially to satisfy the user requirements based on their input parameters. Since decentralised environments comprise large number of distributed repositories with massive stored services, it further increases the complexities of the service discovery process. Therefore, realising the most appropriate services for user enquiry is often complicated in decentralised environments.

Various research works are being carried out[21][31]–[38] and several indexing and searching solutions have been proposed to accelerate the resource retrieval processor [36]–[38]. However, most of the existing methodologies lose

efficiency while addressing the matter in a decentralised environment comprising massive services stored in large number of distributed repositories. Therfore, efficient service discovery in a decentralised environment is still an ongoing challenge especially for the latency-sensitive services. The objectives of this thesis is to develop an efficient indexing and searching model for service discovery in decentralised environments in order to enhance the efficiency of service retrieval process.

1.3 Research Problems Statement

The research problem encompasses the following four research questions.

① Determine how to conduct an in-depth descriptive analytics on the characteristics of various services to propose generic service; service retrieval and service discovery definitions which can coexist with other definitions.

② Determine how to **develop an efficient indexing model** to increase the efficiency of service discovery for service repositories in decentralised environments with massive stored services.

③ Determine how to **develop an optimised searching algorithm** to achieve efficient service discovery in decentralised environments that comprise large number of distributed service repositories.

④ Determine how to **design a suitable simulation environment** to evaluate the proposed efficient service discovery approach in decentralised environments.

1.4 Research Aims and Objectives

This research aims at developing a novel efficient indexing and searching

model for service discovery in decentralised environments with large number of service repositories and massive stored services to reduce traversed services and traversed repositories and therefore retrieval time; ultimately to improve efficiency of service discovery.

In accomplishing this research aims, the objectives of this research study include the following:

① To analyze the existing service discovery techniques and identify the existing research gap towards an efficient service discovery approach in decentralised environments.

② To **develop a novel efficient indexing model** for all service repositories storing massive services in decentralised environments to reduce the number of traversed services and to achieve more efficient service discovery based on analysing the intrinsic characteristics of stored services.

③ To **develop an optimised searching algorithm** in order to efficiently discover services in decentralised environments comprising large number of service repositories based on an optimal routing mechanism and routing index.

④ To **develop a new simulation environment** and to evaluate the proposed indexing and searching model for service discovery in a decentralised environment comprising lager number of repositories with massive stored services.

1.5 Major Research Contributions

This study makes contributions in the area of indexing and searching model for service discovery in decentralised environments and also includes a simulation

environment for experimental validation. The potential contributions of this research work are:

① This thesis puts forward a series of definitions including service, service retrieval and service discovery which can unify the characteristics of services to achieve convenient service computing.

② This thesis propose a novel distributed multilevel indexing model (DM-index) based on one fundamental discrete mathesmatics theory: equivalence theory, which can eliminate embedded redundancy information and narrow the searching space for service repositories containing massive services. The unique multilevel strategy of this index and the equivalence partition can guarantee that each service can only be classified into one subset and can be retrieved efficiently with fewer traversed services.

③ An optimised searching algorithm with double-layer routing mechanism and no-redundancy enhanced bi-direction routing index (DNEB-Chord) has been proposed, which can discover the requested services and locate the destination repository in decentralised environments with fewer hop counts.

④ A new simulation environment, named Distributed Multilevel Di-direction Simulator (DMBSim), has been developed using C# language to evaluate the proposed indexing and searching model for service discovery in decentralised environments.

1.6 Thesis Organisation

The remainder of the thesis is structured as follows.

Chapter 2 presents a detailed survey of the state-of-the-art methodologies in improving service discovery efficiency in order to identify the research gaps and critical issues in the existing techniques for the decentralised environment. Further, this chapter includes an in-detail review of relevant literature related to Objective ①.

Chapter 3 presents the novel efficient distributed multilevel indexing model related to Objective ② including eliminating redundancy; shrinking searching space; creating subset; establishing multilevel index and theoretical validation.

Chapter 4 presents an optimised searching algorithm related to Objective ③ including no-redundancy enhanced bi-direction routing index and double-layer routing mechanism.

Chapter 5 presents an experimental platform as Objective ④ and evaluates the proposed efficient indexing and searching model for service discovery against the sequential index, inverted index and traditional Chord models, in terms of the traversed services, hop counts and retrieval time. The obtained results are discussed in detail.

Chapter 6 summarises the contributions of this research. The remarks and conclusions present the overall the accomplishment of the thesis through an evaluation of the entire research study. Finally, the study's limitations have been discussed along with a discussion of future research.

2 Literature Review

This chapter presents a survey of service computing together with other relevant literature including indexing model, Distributed Hash Tables (DHTs) lookup algorithms and simulation environments which are pertinent to this research work. After reviewing the indexing models, DHTs algorithms and simulation environments, this thesis critically analyzed sequential index, B-tree index, inverted index, Chord, Pastry, Tapestry, Kademlia, CAN, Peersim, OverSim, P2PSim and PlanerSim, then aimed at improving the service discovery performances for the decentralised environments.

2.1 Service Orient Architecture

Service-oriented architectures (SOA) is an emerging approach that addresses the requirements of loosely coupled, standards-based and protocol independent distributed computing [4][39]. In an SOA, all the functions are defined as "services" and are separate from the state or context of others. Generally, a service in SOA consists three essential properties [40]:

a) Autonomous: a given service only maintains its own state.

b) Platform independent: the interface contract to the service is limited to platform independent assertions.

c) Services can be dynamically allocated, invoked and recombined.

Web services are currently the most promising Service-oriented computing (SOC) based technology for realising the objectives in terms of maximum service sharing, reuse and interloper ability [4][6][41]. They use the Internet as the communication medium and open Internet-based standards including the

Simple Object Access Protocol (SOAP) for transmitting data, the Web Services Description Language (WSDL) for defining services and the Business Process Execution Language for Web Services (BPEL4WS) facilitates an orchestration of services in SOA [41]. The SOA and Web services solutions support two key roles: the service requester (client) and the service provider, communicating via service registry [4][42].

2.2 Service Computing

The goal of Services Computing is to enable IT services and computing technology to perform business services more efficiently and effectively In broad terms, service computing including service discovery, service composition, service selection and service retrieval. Service retrieval is a function that accepts a set of parameters and returns a set of appropriate services that can be invoked based on the input parameters set. In general, service retrieval relies on service discovery and composition when searching services from a service repository according to a given parameter set.

2.2.1 Service Discovery

Service discovery is the process of finding one or more services which can satisfy the user requirements. One of the challenges in the service discovery process is the identification of the requested services in different environments. Service discovery involves three inter-related phases: matching, assessment and selection. In the first step, the service description supplied by the developer is matched to the set of available resources. The result (typically a set of ranked web services) is then assessed and filtered by a given set of criteria such as input parameters set. Finally, appropriate services are chosen for subsequent customising and combining with others in service selection [43].

Rambold et al. [44] and Crasso et al. [45] proposed different methods to classify service discovery based on various classification criteria. Based on the service registry, Rambold, et al. [44] classified service discovery into two broad categories such as centralised and decentralised registry architectures [43][46][47]. Figure 2.1 represents typical centralised architecture. It consists of three entities such as service providers which creates and publishes services, service register which maintains a centralised registry of published services and supports service discovery and service requesters which searches the service broker's registries [45].

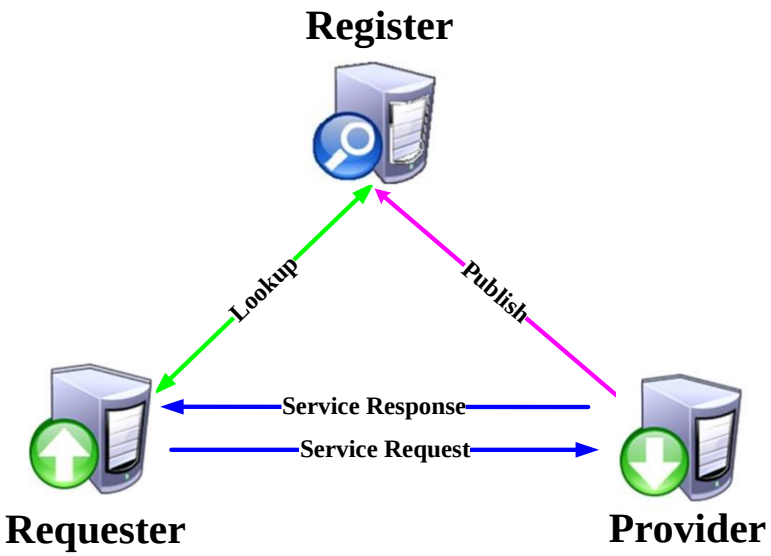

Figure 2.1 Centralised Architecture

Li et al. [46] presented a typical distributed registry architecture, called Web Services Oriented Peer-to-Peer (WSOP), based on the integration of different peer-to-peer systems and Common Web Service (CWS) with SOAP connectivity. In this approach, peers residing as the neighbours are pulled together to form a peer group and at least one peer acts as a super peer to assist a Local Service Registry Broker (LSRB) which can provide a faster service registration and invocation in the peer group environment [39]. The CWS,

hosted on the SOAP server, consists of the service provider, service requestor and the Common Service Registry Broker (CSRB) [46]. The CSRB provides access to nodes located in different physical networks and maintains the mappings of service descriptions between itself and LSRB [39]. In the distributed registry model, if the requested service is not identified in its own service registry, service discovery checks the next registry by calling CWS. The architecture of WSOP is presented in Figure 2.2.

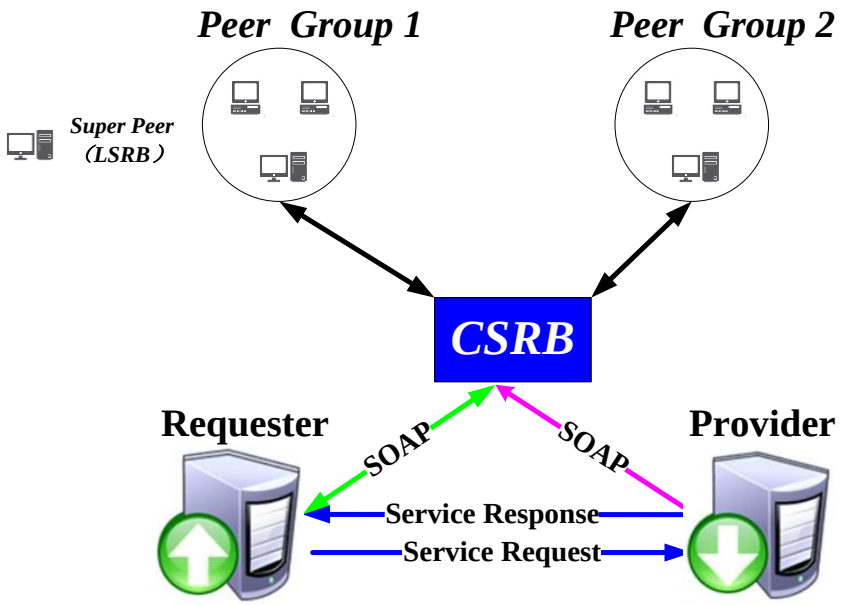

Figure 2.2 WSOP Distributed Architecture

Decentralised service registry incorporates the characteristics of both the centralised service registry and the distributed service registry. Li et al. [48] introduced an agent-based service discovery mechanism (ABSDM), where each node providing web services acts an agent and the WSDL documents are distributed to the other nodes. In this approach, a web-service searching tree has been used which is composed of agents from which a requested service can be discovered.

2.2.2 Service Selection

Service selection is the process of selecting the most suitable services, among those made available by the service discovery process, which can satisfy user requirements. Quality of Service (QoS) is an important parameter that reflects the quality and success of the service selection process. Ran [49] categorized QoS into the following groupes: QoS related to runtime, transaction support, configuration management and cost and security. Runtime related QoS consists of scalability, capacity, reliability, availability, robustness/flexibility, exception handling and accuracy. Transaction support related QoS includes integrity, atomicity, consistency, isolation and durability. Regulatory supported standard, stability/change cycle and completeness are main components of the configuration management and cost related QoS. Security-related QoS is composed of authentication, authorisation, confidentiality, accountability, data encryption, non-repudiation, traceability and audibility.

Service selection is classified into three strategies by Han, et al. [50] as local selection strategy, global selection strategy and mixed strategy. The local selection strategy selects a single and appropriate service from each group of service candidates independently in regards to the other groups. Lamparter et al. [51] proposed a preference based selection approach, which can overcome the issues of random selection of services approach. This approach combines declarative logic-based matching rules based on the weight of the maximum request for each service configuration. Zeng et al. [52] defined a five-dimensional model of service quality and proposed a global optimum selection algorithm based on integer programming. Alrifai [53] broke the global QoS constraints into local QoS constraints and proposed a solution that combines global optimisation with local selection techniques to exploit the benefits of both the search criteria.

2.2.3 Service Composition

Service composition is the process of finding the number of services that can be executed in order to meet given user requirements. Realising suitable composition of services in a large-scale repository for a given request is a complex task in general. Narayanan and McIlraith [54] used Petri nets to analyse the complexity of a composition problem of semantic web services described by DARPA agent markup language for services (DAML-S) and addressed the composition problem. Nam et al. [55] studied the computational complexity of behavioural description-based web service composition and concluded that the composition problem of non-deterministic web services is incomplete and as 2-EXP-complete. Tang et al. [56] introduced a novel automatic web service composition method based on logical inference of Horn clauses and Petri nets. By transforming the web service composition problem into a logical inference problem of Horn clauses based on the forward-chaining algorithm this work further used Petri nets and its structural analysis techniques to obtain the composite service. An increased number of services stored in the service repository leads to a generation of a vast number of rules, which eventually cause the Petri nets of a Horn clause set to be substantial. To reduce the composition time, this work proposed a method to select the candidate clauses for the inference when a new query arrives. However, the drawback of this approach is that it can only be executed after receiving the user requirements, not beforehand.

Wu and Khoury [57] proposed a tree-based search algorithm for web service composition in a Cloud Computing platform. They first created a tree that represents all possible composition solutions according to the user requirements. Then the illegal branches are pruned in order to reduce the response time for the purpose of improving the search performance. Finally, a heuristic algorithm is

used to achieve an optimal search solution. However, the optimisation process of this approach cannot be executed before receiving the user requirements. Constantinescu et al. [58] proposed a type-compatible service composition method, using a forward composition algorithm to obtain an optimal solution. However, the drawback of this approach is that it often contains many useless services in the recommended solution. Kwon et al. [59] proposed a two-phase composition method to overcome such a drawback. Lee et al. [60] proposed a scalable and efficient web service composition method based on a relational database, using service net as a fundamental data structure. In general, service net has two shortcomings. Firstly, it does not consider facilitating service discovery based on the user defined priority. Secondly, it is time-consuming for the service addition and deletion processes.

In practice, different services may be combined together based on their input and output parameter. An example of service composition is illustrated[60] in Figure 2.3, which comprises two different services as described below:

a) A hotel lookup service that accepts destination and date as input parameters and outputs the hotel name, address, telephone number and zip code.

b) A restaurant lookup service that accepts zip code and cuisine as input parameters and outputs the restaurant name, address, telephone number and opening time.

When users want to book a hotel and reserve seats in a restaurant then find the best possible routes between these two destinations, they only need provide the destination, date and cuisine as input parameters. In this example, the navigation system does not accept the input parameters provided by the users as they are not sufficient to determine the travel route. However, the navigation

service from the hotel to a given restaurant can still be accomplished by the composition of the hotel booking service and the restaurant booking service. Service inputs and outputs are indispensable for service composition.

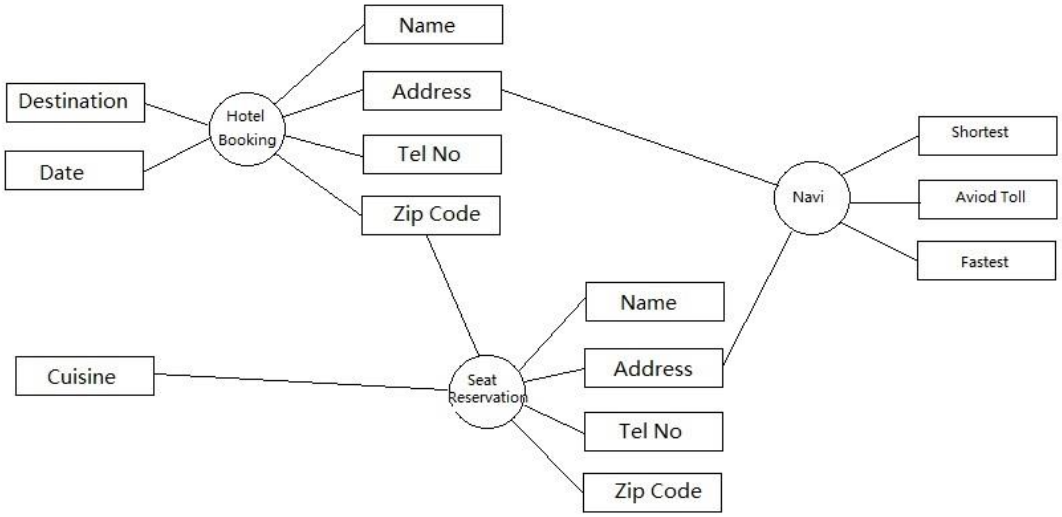

Figure 2.3 Example of Service Composition

2.3 Service Computing Models

In broader terms, service computing can be classified into two categories such as black box and white box templates. In the black, the internal processes of the services are invisible in the box model, which means that the services can only be linked to the input and output parameters. In contrast, the internal processes of the white box model are more visible; all the internal messages should be considered before coupling. AI planning and Petri nets are the two typical representatives for black and white box templates.

2.3.1 Black Box Operation Model

Many different black box models have been proposed in various research papers [59][61][62]. State Calculus is a formalisation model of AI planning, where

every service is abstracted and viewed as an action. The main idea behind this model is using different measures to satisfy user requirements. Meanwhile, an agent is also used to reason service discovery and service composition. Recently, Xu et al. [61] investigated the detection of feature interactions in web services during runtime and proposed a novel detection method. Medjahed et al. [62] proposed an ontology-based framework for the automatic composition of web services, which can generate composite services from high-level declarative descriptions. Furthermore, this thesis proposes an approach for the automatic composition of web services, which consists of four conceptually separated phases such as specification, matchmaking, selection and generation. Kwon et al. [59] proposed a redundancy-free web services composition search method based on a two-phase algorithm. In the forward phase, candidate compositions are computed efficiently by searching the link index. In the backward phase, redundancy-free web services compositions are generated from the candidate composition using the concepts of tokens.

2.3.2 White Box Operation Model

Petri net is a typical representative of the white box model. Petri nets are well-founded process modelling techniques with formal semantics. They have been widely used to model and analyse several types of processes including protocols, manufacturing systems and business processes. Hamadi [63] proposed a Petri net-based algebra, which is used to model control flows, as a necessary component of a reliable web service composition process. This algebra is expressive enough to capture the semantics of complex web service combinations. Liu et al. [64] defined a class of Petri nets called interactive Petri nets to model the search systems, which can be used to analyse the behaviour, find potential problems and improve the searching designs.

2.4 Paradigms of Centralised and Decentralised Environments

2.4.1 Cloud Computing

With the continuing growth of new thchnologies in many dimensions, computing resources are widely spread in an unprecedented scale over the recent years. A new solution named Could Computing (CC) that provides Soft as a service (SaaS), Plaform as a service (PaaS) and Infrastructure as a service (IaaS) on demand, believed to be the future of computing and service. Users are allowed to gain access to remote computing resources that are actually not owned by them [14]. Nowadays, CC has been widely adopted especially for those applications that require rapid deployment of high cost and low use hardware and software resources. There are many descriptions or definitions of CC, but the most commonly quoted one is from the US Department of Commerce National Institute of Standards and Technology (NIST) [65].

Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources e.g., networks, servers, storage, applications and service that can rapidly provisioned and released with minimal management effort or service provider interaction [65].

Cloud services can be categorised into three service groups such as:

① Software as a Service (SaaS) where users can use the cloud provider's applications running on cloud infrastructure.

② Platform as a Service (PaaS) which is the capability provided to users enabling them to create and deploy applications using programming languages, libraries, services and tools provided in the Cloud.

③ Infrastructure as a Service (IaaS), where users have the ability to provision process, storage, networks and other computing resources which can be deployed to execute their applications [65][66].

The conventional CC paradigm is known to be a robust solution for delivering a range of services to users. But in practice, the extended network distance between users and datacenters hinders the provision of real-time service to the application, resulting in a high delay sensitivity [67].

2.4.2 Peer-to-Peer Network

Peer-to-Peer (P2P) network is a distributed networking system where peers (nodes) employ distributed resources to perform networking functionalities in a decentralised fashion. P2P networks can be classified as structured P2P networks and unstructured P2P networks based on the control over the data location and network topology. In the P2P network organisation model, resources can be shared between clients directly without a centralised service management. Nodes in a P2P network play equal roles, in such a way that nodes not only can act as network resources for a single client but also can act as servers for others. Distributed peer networks feature contains both consistent services and open characteristics. Thus, web services are blended with the P2P technology to form a more efficient search application model [68][69]. For instance, Verma et al. [70] proposed a framework based on P2P technology to develop a scalable, efficient, distributed service publishing and discovery model, called METEOR-SWSDI (METEOR-S Web Service Discovery Infrastructure).

In contrast to traditional centralised client-server approaches, P2P networks offer many advantages such as scalability, self-management, self-organisation, fault tolerance and redundancy. These characteristics make distributed system more attractive compared with a centralised system. P2P networks, as a typical

self-organising distributed system, which can eliminate the need for powerful central servers and can locate a set of resources rapidly to overcome failures and performance bottlenecks of a centralised system. P2P can be realised and deployed as overlay networks relying on a physical connectivity among the participating nodes because the peers in the P2P system have equal or similar responsibilities. P2P networks represent self-organised distribution of sets of resources among a set of equal peers for the purpose of enabling subsequent fast lookup of resources [71]. File sharing and storage systems are typical applications which started experiencing the benefits of using P2P architecture domains [72].

Gnutella [73] is the first P2P file-sharing system with a distributed search mechanism, which does not have a centralised directory and does not have any strict control over the network topology or resource placement. Gnutella [73] uses an unstructured overlay network topology that is largely unconstrained and employs flooding approach. Unstructured P2P networks are widely used to share and store documents because of their excellent naturalising properties; i:e for instance, easy addition and updating of resources. However, search queries have to be flooded across the network, which may include problems such as incurring loops in the search process resulting in higher level of message overheads or may return inconsistent and irrelevant results to the search criteria [74].

In structured P2P networks, topology is highly organised have items are placed at specified locations based on Distributed Hash Tables (DHTs), which can address complex queries [75]. Compared to unstructured P2P networks, each peer of structured P2P networks is only responsible for a portion of references to other peers because of the DHTs. As peers and data items mapping

information.is stored in a specific address, each peer can route and lead to the destinat peer that holds the data items by mapping [74].

2.4.3 Fog Computing

The rapid increase in the number of ubiquitous mobile and sensing devices connected to the Internet challenges the traditional network architecture of the CC framework [21]. To address the delay issues of CC for latency sensitive applications, Cisco created the Fog Computing (FC) model as a complementary paradigm to the conventional CC model. FC can extend CC by moving the computation and data storage to the edge of the network which can reduce the additional latency and response delay jitters of latency-sensitive applications [67]. The services in the fog are hosted on the set-top box and access points in the vicinity of end users eliminate the dispensable network hops, thus minimising the response time for applications. Also, FC is a distributed computing paradigm, which empowers the network devices at different hierarchical levels with better computational and storage capability [22][76]. FC is a distributed paradigm that provides cloud-like services to the network edge. In FC, fog nodes can share the unused computing resources for processing applications, or participate in a CC system, or work in a stand-alone mode [77]. It can leverage Cloud and edge resources along with its infrastructure [16][67][78]–[80].

2.5 Indexing Models

With the continuously increasing data volume, variety and velocity of Internet, the ways of handling data have taken new dimensions, leading to the emergence of the big data era. The unprecedented amounts of data generation are now challenging the traditional storage and access mechanisms of both the centralised and decentralised systems. The method of constructing an index to

propose a given database is often used to solve this problem [81]. An index is a collection of information published by the service providers. The index models are usually not authoritative and the information is not centrally controlled. In general, an index may also include outdated information and thus should be subjected to verification before actual utilisation for service discovery. However, addressing an effective way of reducing the space consumed by this index is a crucial requirement [81].

2.5.1 Sequential Index

The sequential index model is also known as non-index. All services are stored in a sequential structure, such as a list. For a retrieval request, all the stored services will be traversed and checked whether they match the given request or not.

2.5.2 B-tree Index

In the past decades, B-tree has been regarded as one of the most popular and efficient index structures amongst the proposed index structures for database systems [82]. Various B-tree access algorithms have been proposed to increase the concurrency in accessing the index and to minimise the data access delay. Although the B-tree index structure and the proposed algorithms have shown to be very efficient for traditional database systems such as banking systems, airline reservation and accounting systems, they have not been successfully applied to database systems requiring stringent response time requirements [83].

2.5.3 Inverted Index

The inverted index is a core data structure of information retrieval systems, especially in search engines [84]. It is regarded as the integral component of almost all retrieval engines, which facilitates easy full-text search since text is the most commonly used data type in communication. It has been widely used in text retrieval services for its merits of less memory consumption, easy management and simple construction. The inverted index consists of a list of terms and a list of postings for each term. Each posting contains the term's information including document ID and term frequency in the document [85]. For example, let us consider the dataset presented in Table 2.1.

Table 2.1 Example of Data Set

ID	Titles
1	Indexing and ranking in databases
2	Keyword search in databases
3	Fuzzy type network and index in databases
4	Networks and network security
5	Security algorithms for network and databases
6	Keyword search and ranking

The keyword 'databases' occurs in documents with IDs 1, 2, 3 and 5. Hence the inverted index for this keyword is stored as follows: firstly, the keyword is stored and then the individual index is stored. If necessary, the total numbers of the index entire are also stored. Table 2.2 illustrates the basic structure of the inverted index for the dataset presented in Table 2.1.

Table 2.2 Example of Inverted Index

Keyword	Inverted Index
Databases	1,2,3,5
Network	3,4,5
Security	4,5
Fuzzy	3
Keyword	2,6
Ranking	1,6

The inverted index dramatically accelerates the process of filtering out irrelevant atomic services. Inverted index promotes the efficiency of the service discovery process by offering better response times than those of the sequential matchmaking method and by providing better recall rate than that when inverted index is not used. Many recent articles have focused on maintainence the inverted index. Aversano et al. [86] proposed a backward composition method containing two steps as horizontal and vertical. The horizontal step is used to identify a minimal set of services that can converge to a target state; and the vertical step is used to repeat the former until the search process meets a stop condition. To reduce the execution time of the service discovery process, Li et al. [87] proposed an inverted index method to manage services which can store services along with parameters. When a parameter is identified to be the output of a particular service, an index link is created from the parameter to the corresponding service. This method is efficient and effective for backward service composition methods and convenient for service addition and deletion. However, it is not applicable to the forward composition methods [57][58][59], which is simpler and more popular than the backward methods. However, the inverted index can also be adapted to the forward composition methods by creating an index link between the parameter and its corresponding input parameter rather than the output service. Still, redundancies are evident in the inverted index in a centralised repository, which can incur additional process

time in the service discovery and composition process. To this end, such redundancies should be rectified in the new decentralised service repositories system.

Inverted index and techniques of compression have been studied extensively in the past [81][84][88]–[91]. In [90], authors suggested using the frequency of each word. This frequency indicates the number of times the indexed word has been searched. The word with the highest frequency count is placed on top of the lookup table. Thus, the words in the table are stored in descending order of their search frequencies. However, this technique has the two drawbacks. Firstly, extra space is needed for storing the frequencies of each word. Secondly, the entries in the table need to be shuffled as and when the frequencies are updated. The frequencies of a word increase, the word is shifted up to a new position as necessary. This shuffling process of entries in the lookup table can create substantial overheads every time the table is updated. A method [91],called signature sorting has been proposed in the works in which the documents are sorted according to abstracts. An abstract is generated by firstly sorting all words in a particular document in descending order of their frequencies. Next, the top n words are chosen as signature vocabulary, where n can be any predefined threshold set by the user. Finally, for each document, a signature is generated by choosing those words belonging to the signature vocabulary and sorting them in descending order of their frequencies. Placing the most frequently searched words at the top of the table can improve the search efficiency. However, this method is suitable for only those scenarios where the word count per document is small and words are repetitive. This strategy also suffers from the disadvantages of consuming extra space for maintaining the frequency counts and the overheads created by constant reordering of words in the lookup table.

In summary, the inverted index structure is a widely adopted model for service management and retrieval in service repositories, but the inverted index includes considerable redundancies which can significantly increase the retrieval time in large-scale repositories. To this end, this research study proposes a new index model for the decentralised systems to enhance service discovery.

2.6 Distributed Hash Tables Protocols

Distributed hash tables (DHTs) are lookup structures that provide a *put (key, value) / get (key)* interface to store and retrieve information, to publish and discover services or devices in a network [92][93]. In DHTs, clients and routers store (key, value) pairs in which services and the location of clients are mapped to keys [94]. In general, the key of a resource is the hash of the particular resource name through which some hashing function is defined by the DHT algorithm, while the value is a short data associated with that resource [95]. DHTs rely on the cooperation of some nodes which collectively provide the information storage and retrieval services. Nodes are arranged in an overlay network, which is built upon an existing network, whose topology depends on the nature of the DHT algorithm. The structure of the DHT topology affects the message routing. Each DHT algorithm typically defines a protocol (usually a set of RPCs) to be used for the communication and cooperation among the DHT nodes [95].

DHTs have been extensively studied through theoretical simulations and analysis [33][35][96]–[98] to enhance the efficiencies of service discovery process by quicker search, fewer query message broadcasting and less resource consumption.

In the recent years, DHTs have become one the hottest and ongoing research issues [99]. Originally, DHTs have been developed for P2P file sharing applications for location substrates. Because of its advantages including excellent routing performance, scalability, fault tolerance, load balancing and decentralisation, DHTs can effectively eliminate the need for directory management of P2P systems. Those properties make DHTs more suitable for deploying services at the application layer [100]. In structured P2P Systems, data is usually distributed among the network nodes. DHTs allow the nodes to effectively identify the peer nodes that contain the specified data item without regarding the location of the querying node [74][101].

Theoretically, most DHTs consist of numerous organised nodes in a well-defined manner [102]. Each node is associated with a unique hash identifier to handle the stored objects. The objects stored by a node can be discovered by other peers nodes later based on the knowledge of the object's hash alone. A given message should have the same hash value despite belonging to different nodes; this can be ensured only when all the nodes in the network follow the DHT protocol correctly. When an object is added onto a DHT, its contents are hashed to produce an identifier which is also mapped into the address space [107]. The originating node then keeps routing the object to the closest-matching neighbour node until no more matching is identified. The last node is considered as the final destination of the object, which usually takes the responsibility of storing the object and any other management. Any individual node can obtain this object based on the knowledge of the object's hash value which is usually the key. In simple terms, the keys are partitioned among the participating peers and messages or queries can be efficiently routed to the sole owner of any given key in a DHT [74]. Of course, this only possible when all the nodes follow the DHT protocol correctly[103].

The following sub-section reviews some of the existing DHTs protocols: e.g. Chord [104], Pastry [105], Tapestry [106], Koorde [107] and Kademlia [108].

2.6.1 Chord

(1) Traditional Chord

Chord [104] is a well-known DHT-based distributed protocol which can locate the peer node containing a particular data item efficiently. The resulting nodes in the overlay are mapped in a circular fashion. A consistent hashing is used to assign objects to the nodes [109]. A lookup process requires $O(log\,N)$ messages in an N-node Chord network. Chord can adapt efficiently to the joining and leaving of nodes in the network using its operations such key location, node addition, node deletion and so on [110]. The Plaxton protocol developed by Plaxton et al. [22] is very similar to Chord protocol. In comparison with the Plaxton protocol, Chord protocol is less complicated and can manage node addition and deletion concurrently.

Chord assigns each node and data (key) with a m-bit ID using a base hash function such as SHA-1. Node's IDs are ordered in an ID circle modulo 2^m with m being the length of the identifiers. The basic principle is to store the data (key)'s value in the first node whose ID is equal to or follows the data in the ID space [33]. Each node in Chord holds a routing table that consists of a finger table entries $1 \leq k \leq m$, called the finger table, where finger [k] indicates the first node on the circle that succeeds $(n + 2^{k-1}) mod 2^m$[33][103][111][112]. A typical finger table [1] of the nodes is shown in Table 2.3.

Table 2.3 Finger Table of Chord

Notation	Definition
Finger [k]	First node on circle succeed $(n + 2^{k-1}) mod 2^m, 1 \leq k \leq m$
Finger [k].ID	Identifier of the k-th node
Finger [k].Address	Address (IP) of the k-th node
successor	The next node on the identifier circle
predecessor	The previous node on the identifier circle

Till now, various schemes to enhance the routing efficiency of the Chord protocol have been proposed by applying various techniques to speed up the searching process of local resources [113]–[115]. The proposed methods can be classed into two categories: the first category speeds up the process of searching resources and the second category matches the topology between the overlay network and the physical network. Some of the newly improved Chord protocols are reviewed and analysed in this section, the primary objectives of most of these improved Chord protocols focused on an efficient and fast lookup of nodes containing the keys.

(2) BNN -Chord

Based on neighbour's neighbour chord, BNN-Chord algorithm has been proposed and developed by F. Chao et al. [114]. By using neighbour's neighbour link based on a learning table, the finger tables are extended in the proposed BNN-Chord algorithm. This extended table can maintain the information of the successor's successor node, which can reasonably increase the finger density of the routing table and helps to find the closest neighbour nodes [116]–[118].

(3) MF- Chord

As another extension to Chord, XM Zhao et al.[115] proposed a new resource indexing model named MF-Chord. The proposed MF-Chord is an efficient P2P system capable of processing multi-attribute multi-keyword fuzzy-matching queries with high recall ratio. Based on the query-frequency, the attributes of the resources can be divided into several sets. Each attribute set corresponds to a bitset of the fingerprint. Main keywords of each attribute are abstracted out and hashed to the bits of the corresponding bit set. Then the fingerprint of the resources which uses fewer bits to reflect the primary keywords information of each attribute is generated and used as the key to map the resource information to other nodes [116].

(4) A-Chord

Based on replacing the redundant information using an external resource in the finger table, C. Cheng et al., [68] proposed an advanced Chord called A-Chord. The redundant information stored in the finger table of Chord ring consumes more valuable space, furthermore, this approach decreases the search efficiency. If all those redundant information are replaced by inspired information [119], each node could establish more links with more nodes using the space occupied by redundant information [116][119].

(5) TB -Chord

TB-Chord, a topology-aware protocol has been proposed in [120], which deals with the routing delay and ignorance of the physical topology information such as IP address. It applies a suite of mechanisms to extend Chord to optimise the utilisation of the physical network topology and the overlay network. In the TB-Chord, nodes are configured with a prefix according to their levelled domain. The related finger tables are set bi-directional and the entry is pointed to the nearest nodes in the particular sphere. The system facilitates the landmark

+RTT method to generate the topology information. For exploiting the topology-aware information in TB-Chord routing design, the landmark +RTT approach has been used to help to build TB Chord's topology based on the geographical location. TB-Chord divides the entire network into k-levelled domains. Nodes in the same level domain insist that the nodes belong to the same physical location [116].

(6) An Improvement to the Chord Based P2P Routing Algorithm (IChord)

D Chen et al., [121] proposed a routing algorithm for the Chord protocol and presented an improvement strategy for the original Chord routing algorithm. When updating the finger table, their algorithm first checks whether there are entries in the table or not. To enlarge the lookup bound and to increase the density of the finger table, the remaining half of the Chord ring can also be added. Here new nodes are selected from the remaining half of the Chord ring according to the ratio of P and Q[122]. P reflects the span of the source node ID n and the largest node ID m. Moreover, Q reflects the degree of information redundancy in the finger table entries [116].

(7) Ant-Chord

Ant-Chord based on the ant colony optimisation algorithm has been proposed in [123], which can resolve the mismatch issues between the physical network topology and the overlay topology. Using the ant colony algorithm, the storage nodes in the Chord ring are regarded as travelling salesman problem. Then the messages obtained from the travelling salesman problem are used to build the Chord. This model is easy to implement since it only incurs marginal changes to the standard Chord model, but it also incurs a few additional overhead costs in the routing table storage [113].

(8) TCS-Chord

TCS-Chord has been proposed based on the Chord scheme [124], which uses one of the selected neighbours to substitute the next relay node. To locate the nodes, a set of coordinate axes is built based on the position of the nodes. Each axis is a machine distributed internet. P2P nodes can ping these axis machines to get their coordinates [113].

2.6.2 Pastry

Similar to Chord [104], Pastry [105] is a fault tolerant DHT-based structured overlay lookup protocol, which organises a network of nodes in a 128-bit circular index space. Given a network of n nodes, a lookup will be completed in $\lfloor log_{2^b} n \rfloor$ hops by deploying Pastry [105], where b is a configurable parameter. To achieve this, each node should maintain a routing table containing $(2^b - 1)\lfloor log_{2^b} n \rfloor + l$ entries (where l is a configurable parameter). Unlike in Chord [104], each node in Pastry [105] has a two-dimensional structure with $\lfloor log_{2^b} n \rfloor$ rows and 2^b columns pulsing l additional leaf entries instead of a one–dimensional index. Each entry in row n points to a node with an ID matching the current node's ID in the first n digits, but with a different $n + 1$ digit. This allows a lookup to be forwarded towards its destination, in such a way that each hop forwards the query to a node whose ID shares at least $n + 1$ digits with the key, compared to the n digits at the previous node [125]. Peer nodes can join and leave the network without causing the crashing network. Pastry governs the nodes to store the key values as a portion of the DHT based on its design. It can also use external routing systems to determine the most efficient routes. Pastry [105] protocol is a prefix-based data location protocol [23]. Although the routing latency can be reduced in Pastry [105] protocol, more complicated addition operation is often required to initialise the routing tables of the newly joined nodes [125]–[128].

2.6.3 Tapestry

Rather than nodes being organised into a ring structure as in Chord [104] and Pastry [105], the overlay of Tapestry [106] is viewed as a mesh-like structure. Each node is seen as the root of a tree with the leaf nodes being pointed in the routing table. However, the routing scheme of Tapestry [106] is similar to Pastry [105], in such a way that a digit is "corrected" in the node ID after each hop. It guarantees that a lookup can be completed in $log_\beta b$(in base-β) hops, again with the assumption that the routing information is correct. Each node in Tapestry [106] holds a neighbour map containing $C \times \beta \times log_\beta n$ entries in a network of n nodes. Additionally, a node stores back pointers to other nodes, which are used for the node joining operations. Each level of the neighbour map represents a partially matching key. Within each level, an entry is selected based on the closeness of a given node to the current node in an attempt to minimise latency. This locality selection is another property shared with Pastry but not Chord [103][104][125].

2.6.4 Kademlia

Kademlia [108] is a DHT-based fully decentralised structured overlay which operates over a fixed-size 160-bit node ID space, using a similar notion to Pastry [105] and Tapestry [106] with parts of the node ID being "fixed" with successive hops through the network. Each node has one k-bucket which is used to maintain nodes within a distance of between 2^i and 2^{i+1} from itself. The distance is defined as the integer representation of the XOR between the two identifiers, which can be used to lookup a specific key. Routing information stored in the k*-buckets* are updated when a node receives a message from another node rather than simply assuming that nodes issuing large number of queries become more popular [125]. To lookup a specific key,

the query originator searches the local routing tables for nodes with the closest distance to the query originator and then contacts them in parallel. Kademlia [108] offers lower lookup latency than Chord [104] since it can select any α nodes from the available *k-bucket* to forward the lookups, rather than being restricted to a single correct node as in the case of Chord [104]. Kademlia [108] has been widely adopted by several popular applications, e.g. BitTorrent. However, Kademlia [108] can compensate the messaging costs of Chord [104] only when $\alpha = 1$, implying no parallelism in the lookup steps [136]-[139].

2.6.5 CAN

CAN [34] has been proposed as a scalable Content Addressable Network. The design of CAN is entirely different from that of Chord, Pastry or Tapestry since the number of neighbours of each node are fixed and also independent of the number of nodes in the network. Keys are mapped to points within a virtual multi-dimensional coordinate space and each node is responsible only for a portion of the virtual space. Each node maintains a routing table containing the addresses of its neighbours in the space [122][135]. A lookup is routed by passing through the neighbouring nodes that are responsible for their respective portions of the coordinate space and the coordinates of the current node to the destination node should form a straight line through the neighbouring nodes [125]. CAN protocol centres around a virtual d-dimensional Cartesian coordinate space on d-torus [24]. Unlike the Chord protocol, there is no linear dependency between the nodes in the CAN network and the network size. However, data location CAN protocol is costlier than that of the Chord protocol.

DHT lookup algorithms such as Chord [104], Pastry [105], Tapestry [106], Koorde [107] and Kademlia [108] have been applied in P2P networks in order

to find the required resources from the host nodes efficiently. Chord [104], is a widely used protocol for service retrieval in structured P2P networks. However, it is not sufficiently abundant for efficient service discovery and retrieval services among all the nodes in the network. To Address this issue, this research study attempts to enhance the node allocation efficiency in a decentralised system based on an optimised Chord algorithm named DNEB-Chord

2.7 Simulation Environments

In this section, some of the existing simulators, such as PeerSim [136], OverSim [137], P2PSim [138] and PlanetSim [139] are reviewed and analysed. PeerSim [136] has been developed as a simulation tool for researchers at University of Bologna and Trento, Italy. OverSim [137] is a flexible overlay network simulation framework based on OMNeT++, which applies to both structured and unstructured P2P networks. P2PSim [138] is a discrete event simulator for structured overlay networks written in C++. PlanetSim [139] is an object-oriented simulation framework for overlay networks and services written in Java.

2.7.1 PeerSim

PeerSim [136] is the only simulation testbed facilitating non-hierarchical implementation of different architectures in peer-to-peer networks. It provides an independent handling of protocols in a real unstructured environment. Later, this simulator was released under the LGPL open source license to make it available for other research projects [136]. Similar to many other peer-to-peer and volunteer computing simulations, PeerSim does not need to instantiate a complete network topology [136]. It has been supported by several P2P protocols, facilitating easy deployments. Furthermore，users can easily control

the implementation of the protocols in PeerSim with simple configurations [136]. These files are edited in plain ASCII text model comprised of key-value pairs representing *java.util* properties [19][110].

However, PeerSim is not very user-friendly since it contains no Graphical User Interface (GUI). Furthermore, PeerSim generates larger amounts of data during simulation, which are difficult to handle [136]. Therefore, it can be concluded that PeerSim is not very good from a usability point of view, but it can be used for writing utterly decentralised peer-to-peer overlay network protocols with high scalability and efficiency based on petite code sizes [136][141]. This makes PeerSim less attractive among the research groups.

2.7.2 OverSim

OverSim [137] facilitates a user-friendly GUI and supports a wide range of both structured and unstructured P2P protocols such as Chord [104], Pastry [105], Tapestry [106] Koorde [107] Kademlia [108] and GIA [142] etc. Such protocols can be used for both simulation and real-world networks [143]. OverSim facilitates the implementation of uncommon and rare P2P protocols with more comparable functionalities, for instance, a generic lookup mechanism for structured peer-to-peer networks and an RPC interface [137][143].

OverSim uses discrete event simulation to stimulate the exchange and processing of network messages [137]. Usually, most of the non-profit users choose the free open source simulation framework OMNeT [137]. OMNeT++ is highly modular and supports the implementation of both the compound modules and direct implementation of simple modules in C++ and the functionalities are defined in a simple definition language called NED. These modules communicate and exchange messages via gate connection [143][144].

2.7.3 P2PSim

P2PSim [138] supports seven peer-to-peer protocol implementation including recent protocols Koorde [107] and Kademlia [108]. Maximising performance concurrency and minimising synchronisation requirements are the two core compositions of P2PSim's [138] control flow design. Additionally, avoiding deadlocks have also been a primary consideration of P2PSim. Further, it supports a network delay model constructed from actual measurements of Internet hosts [2]. But, due to its design trade-off towards latency, this simulator scales well only for experiments where latency is the bottleneck instead of queueing delay, bandwidth, or packet loss. Compared to real implementations, protocol code written for P2PSim is easier to understand, since the code is significantly fewer. Similar to other discrete event simulators, P2PSim can primarily be used to compare, evaluate and explore P2P protocols [138]. Without influencing the performance, P2PSim supports easy configuration and comparison of various protocols with marginal changes to the source code. For real implementation, pseudo-code can be used to write programs for P2PSim with ease [138].

2.7.4 PlanetSim

PlanetSim [139] can support both structured and unstructured P2P overlay networks. Without the consideration of bandwidth and latency costs, PlanetSim has an apparent underlying network layer, which limits the availability of the simulation statistics. Therefore, PlanetSim is not very challenging and unsuitable to simulate heterogeneous access networks and terminal mobility. It is possible to visualise the overlay topology at the end of a simulation run, but there is no interactive GUI [137]. The layered architecture of PlanetSim is built

across three hierarchical layers such as application, overlay and network layer [145].

With its well-structured design, a clean API is available for the implementation of overlay algorithms and application services. Users can simulate a request layer service with different overlay algorithms [139]. It distinguishes between the creation and validation of the overlay algorithms and the creation and testing of new services on the top of the existing overlays. Also, PlanetSim provides an Ant algorithm which can intelligently add new links on the overloaded paths within the P2P overlay and hence improves the P2P overlay routing performance [145]. PlanetSim has an excellent and very clear hierarchy API and provides proper extension interfaces for entities. Also, an existing entity can easily be replaced to deploy the implementation as per the simulation settings. Furthermore, I Baumgart, B Heep and S Krause stated [145] that PlanetSim could provide some basic statistics such as total simulation time through to in-depth simulation statistics through aspect-oriented programming (AOP) [139].

2.8 Summary

To conclude, this chapter presented a detailed analysis of service computation, distributed networks, DHTs lookup algorithms and a few existing P2P simulators. Various service discovery methodologies have been recently proposed and various enhancements and/or modifications to resource management have been suggested for effectively exploiting the service repository. The inverted index is one of the widely used efficient indexing models, but it incurs a significant proportion of redundancy information. Effectively reducing this redundancy information for achieving quicker query resolution is still an open challenge for efficient service maintenance. With this

in mind, the next chapter proposes the novel efficient multilevel indexing model based on equivalence theory to enhance the service retrieval efficiency.

3 Distributed Multilevel Index Model

This chapter presents a novel multilevel indexing model for service repositories storing massive services in decentralised environments based on equivalence theory. The proposed indexing model reduces the number of traversed services and can effectively resolve a retrieval query by eliminating redundancy information and narrowing searching space.

3.1 Definitions

Before deliving into the description of the proposed indexing model, several basic definitions and theories should be introduced and analysed. A series of definitions should be considered for carrying out further work in this research study which helps to sum up the essential characteristics of massive stored services. These definitions should also be compatible in the context of indexing and searching models in terms of versatility, compatibility, extensibility and upgradeability. Many service definitions have been in literate previous research works, one of them is shown in *Definition 3.1* [56][59][87][146].

Definition 3-1: *Service Definition 1*

$$s = (I, O)$$

a) s symbolises service,

b) I represents the input parameters set,

c) O represents the output parameters set.

As more parameters should be considered in practice, a service s is defined fully as a 4-tuple composition, introduced by Tang et al. [56], as in *Definition 3.2*.

Definition 3-2: *Service Definition 2*

$$s = (I, O, C, QoS)$$

a) I represents the set of semantic concepts referenced by the input parameters,

b) O represents the set of semantic concepts referenced by the output parameters,

c) C represents the set of behavioral constraints which are the conditions imposed by the service provider on s to assure its correct execution,

d) QoS represents the quality attributes of service s, such as cost, response time, availability and reliability.

Bartalos and Bielikova [146] postulated another perspective of service definition, as shown in *Definition 3.3*.

Definition 3-3: *Service Definition 3*

$$s = (I, O, Pre, Post)$$

a) I and O represent the lists of input and output parameters sets,

b) Pre represents the condition that should be met before service execution,

c) $Post$ represents the condition after service execution.

From *Definitions 3.1, 3.2* and *3.3*, it is clear that service is represented by its input and output parameters and related attributes. After conducting an in-depth descriptive analysis on the characteristrics of services, this thesis adopts a 3-tuple composition for service definition as shown in *Definition 3.4*.

Definition 3-4: *Service Definition*

$$s = (P_{si}, P_{so}, Q)$$

a) P_{si} represent the input parameters set,

b) P_{so} represent the output parameters set,

c) Q represents all the remaining attributes of service cover all the prior descriptions including $Pre, Post, BC$ and QoS etc.

As mentioned in Chapter 2, service retrieval is an act of finding a subset of services matching the user requirements, which can be invoked by service composition and discovery. However, in each step of service composition, services are retrieved and inspected to see whether or not they can be invoked . In a large-scale service repository with massive stored services, most of the services are irrelevant and unnecessary to be retrieved. To address this issue, it is necessary to unify the definitions of user requests, service discovery and service retrieval in this research study. User requests can be denoted as in *Definition 3.5*.

Definition 3-5: *User Requests*

$$Q = (Q_p, Q_r)$$

a) Q_p represents the parameters set provided by user,

b) Q_r represents the parameters set required by user.

In this research study, service retrieval is defined as an operation that accepts a set of parameters and returns a set of services that can be invoked by accepted parameters set. For a given request Q, service retrieval can be defined as in *Definition 3.6*.

Definition 3-6: *Service Retrieval*

$$R(Q_p, S) = \{s | (P_{si} \subseteq Q_p) \wedge (s \in S)\}$$

a) Q_p represents the parameters set provided by user,

b) S represents the service set,

c) s represents the retrieved service,

d) P_{si} represents the input parameters set of retrieved service s.

e) $\wedge$ represents basic mathsmetics function conjunction

Furthermore, service discovery can be defined as in *Definition 3.7*.

Definition 3-7: *Service Discovery*

$$D(Q, L, S) = \{s | (P_{si} \subseteq Q_p) \wedge (Q_r \subseteq P_{so}) \wedge s \angle L \wedge s \in S\}$$

$$= \{s | R(Q_p, S) \wedge (Q_r \subseteq P_{so}) \wedge s \angle L\}$$

a) Q represents user requests,

b) Q_p represents the parameters set provided by user,

c) Q_r represents the parameters set required by user,

d) L represents the set of constraints that must be met,

e) S represents the service set,

f) s represents the discovered service,

g) P_{si} represents the input parameters set of discovered service s,

h) P_{so} represents the output parameters set of discovered service s,

i) $\wedge$ represents basic mathsmetics function conjunction,

j) $s \angle L$ represents that discovered service can satisfy the constraints.

Comparing *Definition 3.6* and *Definition 3.7*, it is clear that service retrieval is an integrated component of service discovery. This means that service discovery can achieve better efficiency if the time of service retrieval is reduced. This is the fundamental reason why this research study mainly focuses on service retrieval instead of service discovery.

3.2 Equivalence Theory

In practice, a single service repository may include [31][32] similar characterising services with the same input and output parameters. Based on the service retrieval definition, a set of invoked services are usually returned according to the input parameters set. A number of services including some irrelevant services may be returned based on a given user request. Thereby it will affect the service composition and discovery efficiencies. If the service retrieval process can reduce the number of invoked services and eventually eliminating the least probable solutions, the number of traversed services for a given query can be significantly reduced. This in turn allows the service discovery methods to search the desired services from a smaller number of service sets instead of searching the entire service sets. In this way of service

retrieval, the process time can be significantly reduced, which can further improve the efficiency of service discovery.

In theory, the search space can be reduced if services with the same parameters are clustered into a virtual subclass. This method is simple but effective for those service sets containing many duplicated services. After this clustering process, a service can be categorised into new virtual service classes. When the composed workflow is completed, the service in the original service repository needs to be blinded. However, this proposal may bring new potential research problems as below.

1. *How can a service avoid to be clustered into two different virtual subclasses with duplication?*

2. *How to guarantee clustering every service into at least one virtual subclass without missing?*

To address the above two issues, an equivalence relation and number of definitions need be introduced which form the core theoretical foundation for the new proposed indexing model. Relationship is a very general definition. In general, ordered pairs are simply a list of elements those are related. For instance, the notation $(a, b) \in A$, can also be denoted as aRb and it simply means that a is in relation with b, and whatever the relation R can be. In mathematics, an equivalence relation may take various perspectives such as a binary relation, a reflexive relation, a symmetric relation and a transitive relation defined as below [147].. Before equivalence theory and relation are introduced, the concepts of reflexive, symmetric and transitive relations are explained in *Definitions 3.8*, *3.9* and *3.10* respectively [147].

Definition 3-8: *Reflexive Relation*

In mathematics, a binary relation R over a set A is reflexive if every element of A is related to itself. Formally, this may be written $\forall a \in A, \ if \ aRa$, then relation $R \ on \ A$ is reflexive relaiton [147]. In other words, every element is in relation with itself.

Definition 3-9: *Symmetric Relation*

In mathematics and other areas, a binary relation R over a set A is symmetric if it holds for all a and b in$A$ that a is related to b if and only if b is related to a. In mathematical notation, this is:

$$\forall a, b \in A, aRb \Leftrightarrow bRa$$

then relation R on A is symmetric [147]. It means that a is related to b, then b is related to a.

Definition 3-10: *Transitive Relation*

In mathematics, a binary relation R over a set A is transitive if whenever an element a is related to an element b and b is related to an element c then a is also related to c. In mathematical notation, this is:

$\forall a, b, c \in A,$ if aRb and bRc implies aRc, then relation of R on A is transitive [147].

In terms of digraphs, reflexivity is the characteristics of including at least a loop on each vertex; symmetry means that any arrow from one vertex to another will always be accompanied by another arrow in the opposite direction; and transitivity insists that there must be a direct arrow from one vertex to another if one can walk from that vertex to the other through a list of arrows, always travelling along the direction of the arrows [147].

In mathematics, an equivalence relation is a binary relation that is at the same time a reflexive relation, a symmetric relation and a transitive relation. The relation "is equal to" is a primary example of an equivalence relation. Thus for any numbers a, b, and c:

$a = a$ (reflexive property),

if $a = b$ then $b = a$ (symmetric property), and

if $a = b$ and $b = c$ then $a = c$ (transitive property).

As one fundamental theory of discrete mathematics, the equivalence relation is defined as *Theorem 3.1* in this thesis [147].

Definition 3-11: *Equivalence Relation and Partition*

Various notations are used to denote that two elements a and b of a set are equivalent with respect to an equivalence relation R; the most common are "a ~ b" and "a $\equiv$ b", which are used when R is implicit, and variations of "a ~R b", "a $\equiv$R b", or "aRb" to specify R explicitly.

As a consequence, an equivalence relation provides a partition of a set into equivalence classes. A partition of set A is a collection of non-empty, pairwise disjoint subclasses which should cover all the components of set A [147]. To prove R has an equivalence relation on a set A, the equivalence classes of R should form a partition of set A. For a given element x in set A, the equivalence class of x to be the set of all elements of A that are equivalent to x, which can be notated as $[X]$. In symbols, $[X] = \{ y \in A | xRy \}$.

Firstly, for $x \in [X]$, it is obvious that no equivalence class is empty.

Secondly, if x and y are in different equivalence sub classes, then $(x, y) \notin R$ and then $[X] \cap [y] = \emptyset$.

Finally, for any elementx ∈ A, it is obvious that x ∈ [X], and so the union of all equivalence classes covers A.

This operation based on equivalence relation is defined as equivalence partition [147]. To support the description of the proposed indexing model, some basic functions including injections, surjections and bijections are formally introduced. In mathematics, injections, surjections and bijections are calsses of functions distinggulised by the manner in which arguments (inout expressions from the domain) and images (output expressions form the codomain) are relaited or mapped to each other[148][149].

Definition 3-12: *Injection Function*

The function is injective (one-to-one) if each element of the codomain is mapped to by at most one element of the domain. An injective function is an injection The function f is injective also called one-one [148]. A function is injective onlt when every possible element of the codomain is mapped to at most one argument [148][149]. Notationally:

f: X → Y, if and only if

∀x, x′ ∈ X, it has $f(x) = f\left(x'\right) \Rightarrow x = x'$

Definition 3-13: *Surjection Function*

The function is surjective (onto) if each element of the codomain is mapped to by at least one element of the domain. A surjective function is a surjection. In other words, every element in the codomain has non-empty preimage [148][149]. Notationally:

f: X → Y, if and only if

$\forall y \in Y$, it has $\exists x \in X \Rightarrow f(x) = y$

Definition 3-14: *Bijection Function*

The function is bijective (one-to-one and onto or one-to-one correspondence) if each element of the codomain is mapped to by exactly one element of the domain. That is, the function is both injective and surjective. A bijective function is a bijection. An injective function need not be surjective, it means that not all elements of the codomain may be associated with arguments. Either a surjective function need not be injective means that some images may be associated with more than one argument. The four possible combinations of injective and surjective features are illustrated in the diagrams to the right [148][149]. Notationally:

$f: X \to Y,$ if and only if

$\forall y \in Y$, it has $\exists! x \in X \Rightarrow f(x) = y$

3.3 Distributed Multilevel Indexing Model

3.3.1 Deployment Mechanism

This research study has systematically reviewed the existing most common indexing methods in service retrieval and discovery. The current service repositories suffer considerable limitations such as limited storage space, excessive redundant information and inconvenient operation architecture. Thus, the operational efficiency of the service discovery is still far from reaching user satisfaction, especially a large-scale decentralised environment comprising larger number of globally distributed service repositories with massive multiple sources services and dynamically changing requirements. To overcome such drawbacks and lacking qualities of the existing service discovery methods, this

research is proposing a novel indexing model and optimised searching algorithm in order to enable efficient service discovery and easy service operations for the decentralised system that consists of numerous repositories with massive stored services. The deployment mechanism of the proposed efficient indexing and searching model for decentralised environments is illustrated in Figure 3.1.

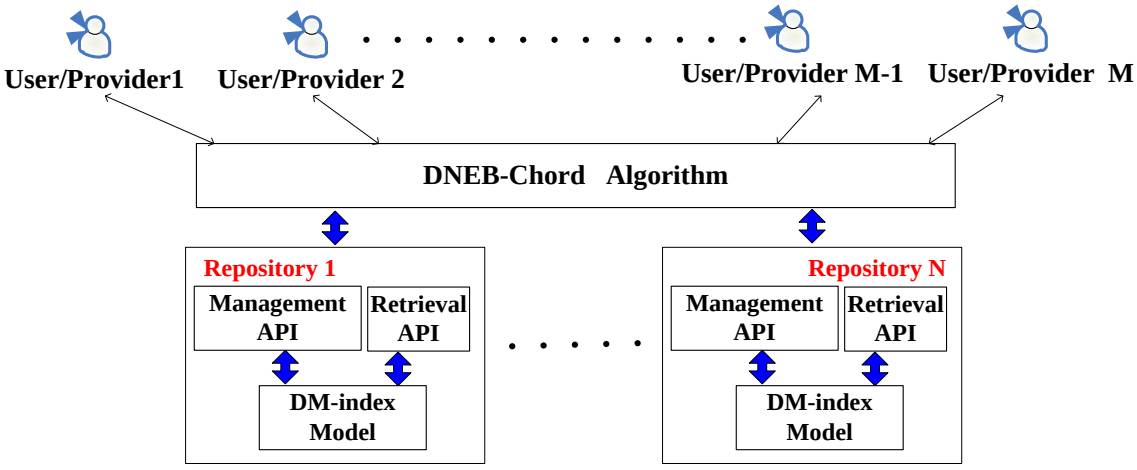

Figure 3.1 Decentralised System with Proposed Indexing and Searching Model

3.3.2 First Level Index

From the works of Y. Wu et al [31][32], services in a repository can be invoked according to the input or output parameters set. In general, there are usually a couple of services associated with the same input and output parameters stored in the service repositories of decentralised environments. It is common that more services than required are usually retrieved to recommend a complete composition of services to users; this can significantly reduce the service discovery efficiencies. A possible way to reduce the search space is to form classes of clusters with the same input and output parameters. Based on the prior *Definitions* and *Theorem*, a set of stored services (S) in repository can be

divided into several subsets by the equivalence relation R_1 and subclass is called as same-class(C_s). Each same-class C_s may contain more services with the same input and output parameters. Thus, each same-class C_s has a unique pair of parameters sets, denoted as P_{csi} and P_{cso}. Therefore, the first level index between same-class C_s and original service set S could be defined as shown in Figure 3.2. From *Theorem 3.1*, each service in original service set S is only indexed by one same-class C_s once, implying that none service being omitted or indexed twice. In other words, C_s set has the integrity and contains no redundancy.

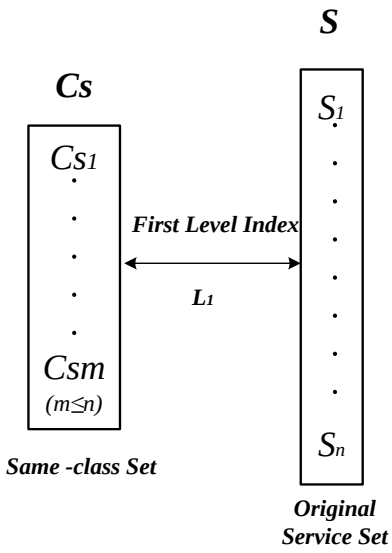

Figure 3.2 First Level Index

The first level index between the original service set S and the same-class C_s set can effectively reduce the redundancy caused by the services with the same parameters set. Without the first level index, all services need to be retrieved for computing $R(Q_p, S)$. With the first level index, only the same-class C_s set needs to be retrieved $R(Q_p, C_s)$ instead of all the services. Since the count of same-class set usually is smaller than that of original service set, the retrieval time is shorter. If there are no or only a few services sharing the same

parameters set, this index would have no or little effect to improve efficiency. Thus in such a case, the deployment of the first level index can be avoided in practice, which will be discussed in the following adaptive deployment section.

Overall, the first level index class between same-class set and original service set not only simplifies the data structure of service repositories but also enhances the search efficiencies.

3.3.3 Second Level Index

The clusters formed with the same-class C_s set in the first level index are further divided into subset classes in this step. Similar to the earlier discussion in the firs level index, R_2 is an equivalence relation on same-class C_s set which is defined as *Definition 3-15* in this thesis.

Definition 3-15: *Equivalence Relation* R_2

$$R_2: \quad C_{sj}, C_{sk} \forall C_s, \qquad C_{sj} R_2 C_{sk} \Leftrightarrow P_{csij} = P_{csik}$$

a) C_{sj} and C_{sk} represent two elements included in same-class set,

b) C_s represents the same-class set,

c) P_{csi} represents the input parameters set of same-class C_s,

d) R_2 represents the equivalence relation on same-class C_s set.

So the same-class C_s set can be divided into several subsets based on the equivalence relation R_2 and each subset is called as input-similar class C_{is} which contains at least one or more same-classes C_s having same input parameters set.

The second level index is postulated to be integrated between C_{is} and C_s in this step as Figure 3.3 indicates. With the help of this second level index, $R(Q_p, S)$ only focuses on input-similar C_{is} set rather than same-class C_s set. The search space can be further narrowed to input-similar C_{is} set in order to improve the efficiency of service discovery and composition.

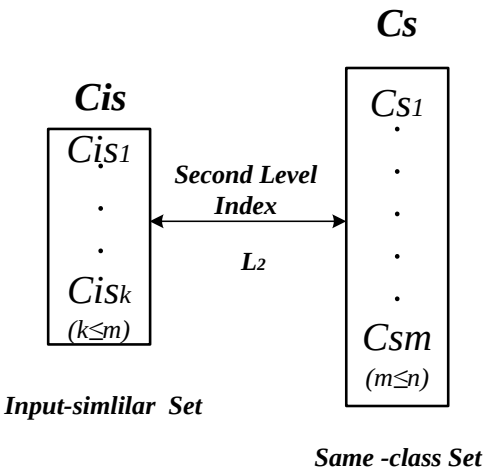

Figure 3.3 Second Level Index

The proposed integration of the first and the second level indexes are shown in Figure 3.4. As the *Theorem 3.2 is* based on the characteries of equivalence theory, the second level index can reduce the redundancy caused by the services with the same input parameters set because the service retrieval process is related only to the service input parameters set. The construction of the first and second indexes do not have any deteriorating effects on the service retrieval process, but they can improve the efficiency of the service discovery and composition. With this structure, the search space could be narrowed into input-similar C_{is} set and the efficiency of service discovery should be further improved. If there are no or only a few services sharing the same input parameters set, the second level index would have no or little effect on efficiency. In this case, the deployment of the second level index can be ignored

[31][32] in practice, which will be discussed in the following adaptive deployment section.

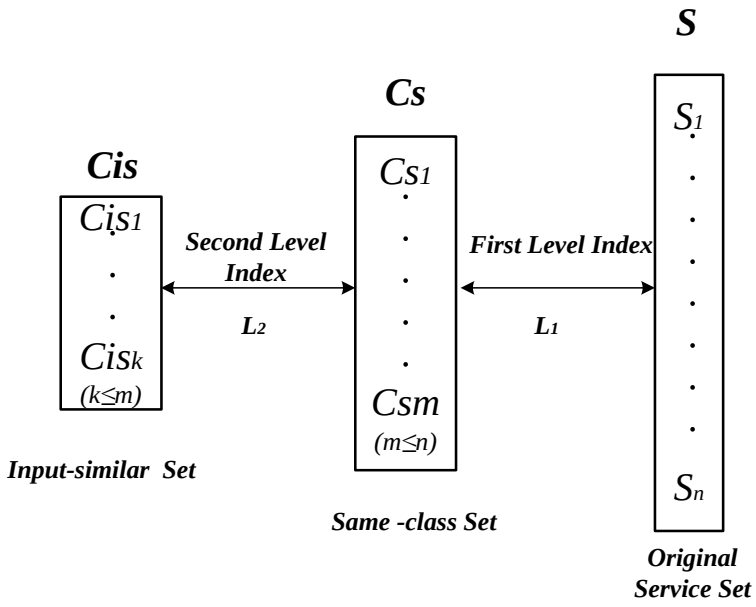

Figure 3.4 First and Second Level Indexes

In sum, the second level index class between C_s and C_{is} does not only simplifies the data structure of the service repositories but also enhances the search efficiencies.

3.3.4 Third Level Index

After the first and second level indexes have been created, the retrieval space could be narrowed. But when computing $R(Q_p, S)$, all the input-similar classes C_{is} need to be retrieved and checked whether $P_{cisi} \subseteq Q_p$ or not. In a large-scale service repository with massive stored services, it is obvious that most of the input-similar classes C_{is} are irrelevant to a single user request. Therefore, in order to further narrow down the search space, it is necessary to form another level index class. The above issue can be abstracted into the following question:

given a Q_p, how efficiently can all the subsets of that set be identified from a pool of service sets.

As reviewed in Chapter 2, B-tree index is an efficient index structure for service retrieval. But it cannot be used to narrow the search space for set comparison directly. For example, there are three general service sets *A*, *B* and *C* which are stored by a B-tree, as shown in Figure 3.5. Since the subset to which the given set $Q_p(a,b)$ belongs to is unknown at this stage, all the three service set *A*, *B* and *C* should be compared with $Q_p(a,b)$, Therefore, the B-tree cannot reduce the search space.

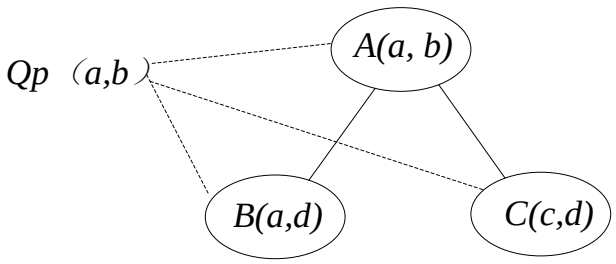

Figure 3.5 Example of Service Retrieval with B-tree Index

As mentioned in Chapter 2, the inverted index is another potential candidate that can reduce the search space. To the same example shown in Figure 3.5, an inverted index would store the subsets as illustrated in Figure 3.6. Now for the requirements of the same set $Q_p(a,b)$, the search space is quickly narrowed to *A* and *B* instead of all the sets because of adapting inverted index.

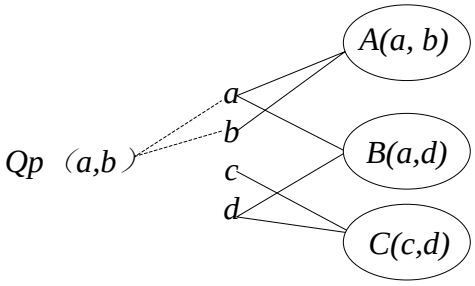

Figure 3.6 Example of Service Retrieval with Inverted Index

Although the inverted index can narrow the searching space, it still contains redundancy. For the proposed input-similar class C_{is} as indicated in Figure 3.7, the inverted index can be deployed between the input parameters and input-similar C_{is} set. For instance, parameters a and b are linked to these input-similar classes C_{is} containing these two parameters. According to parameter a, C_{is1} and C_{is2} are retrieved; according to parameter b, C_{is1} is retrieved. It is obvious that C_{is1} is retrieved twice, which is redundant.

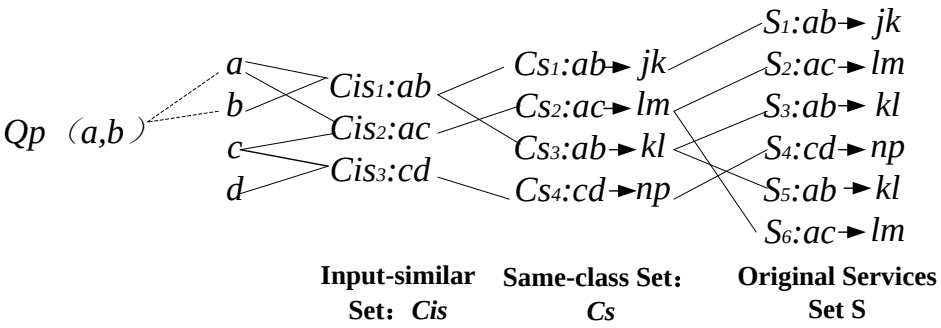

Figure 3.7 Example of Service Retrieval for Input-similar Classes with Inverted Index

To service retrieval request $R(Q_p, S)$, only C_{is1} and C_{is2} linked by the input parameters *a* and *b* are retrieved to check whether satisfied or not, instead of all the input-similar classes. Therefore, the search space of a large-scale repository could be significantly narrowed. For the reason that the input-similar set could not be used directly, a new index level such as B-tree index or inverted index

should be introduced. However, just as discussed in Figure 3.5 and Figure 3.7 that either B-tree or inverted index cannot eliminate all the redundancy. Clearly, it is unnecessary and such redundancies should be eliminated. The elimination of such redundancies will be discussed in the following section. Therefore, constructing the third level index for the decentralised environment will be one of the key challenges in the proposed indexing model.

To resolve this problem, both characters the characters of services and DHTs should be considered as one unit to propose the third level index. In this research study, each service will have a unique hash ID and the service should be guaranteed for storage in the right repository based on the hash matching. In this research study, the randomly selected input parameter ($I_{rs} \in P_{si}$) for upcoming service addition opeartion will be chosen as key for each service to establish the third and fourth level indexes.

In detail, when adding a new service to the decentralised system, the destination repository for the new service is usually decided by the hash mapping. One of the input parameters ($I_r \in P_{si}$) will be randomly selected for generating the service hash ID. An optimised searching algorithm will be developed in Chapter 4 which can provide efficient service discovery across distributed repositories. Based on the equivalence relations R_1 and R_2, the randomly selected input parameter (I_r) should satisfy relevant same-class $I_r \in P_{csi}$ and input-similar class $I_r \in P_{cisi}$ respectively. Therefore, input-similar C_{is} set can further be divided into many subsets by the randomly selected parameters and a new set in each service repository including all the randomly selected input parameters is named as P_{rs} set. Since various services might be added to the same repository with same randomly selected input parameter, each element of P_{rs} set should be indexed to at least one input-similar class C_{is}. Therefore a new level index can be established between P_{rs} set and input-similar set. The

third level index is presented in Figure 3.8 and the overlay view of all the three level of indexes developed is shown in Figure 3.9. Although, the establishing progress of the third level index is different from the first and second level indexes, all elements of C_{is} could be mapped by parameters in P_{rs} set.

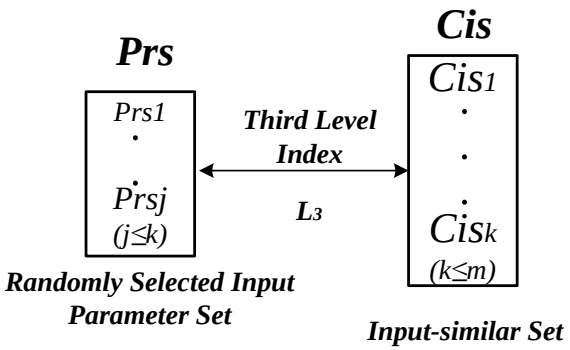

Figure 3.8 Third Level Index

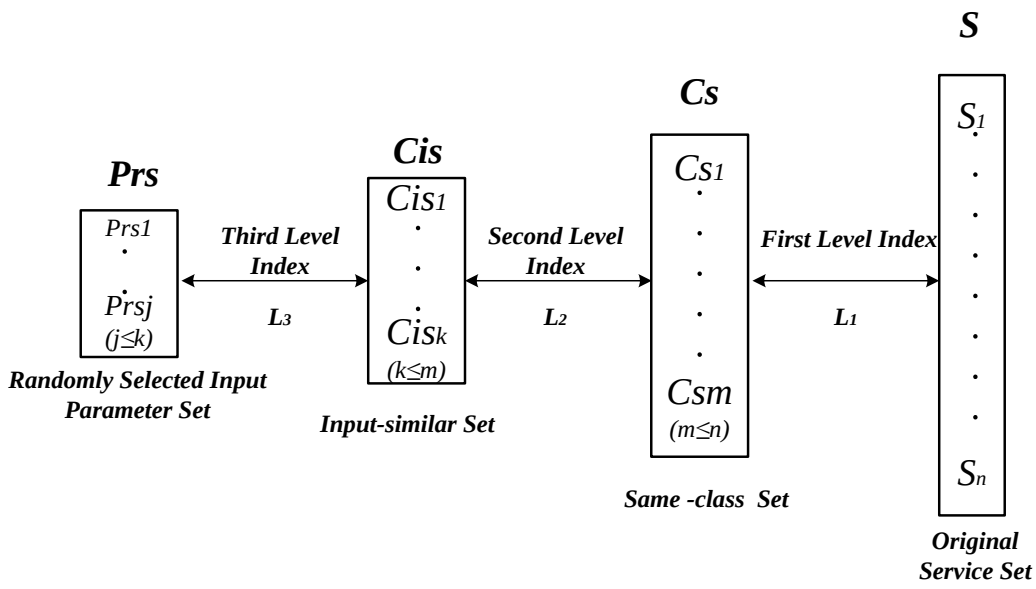

Figure 3.9 First, Second and Third Level Indexes

3.3.5 Fourth Level Index

The developed third level index may not be used for retrieval directly. Thus a fourth level, inverted index, is structured between the all the randomly selected input parameters I_r and P_{rs} set. Since each randomly selected input parameter I_r only can index one element in the P_{rs} set. The *Definition 3.14* of bijection ensures integrity and contains no redundancy, so redundancy in the proposed fourth level index (inverted index) is eliminated. The complete structure of the developed multilevel indexing model, named DM-index, for the repositories with massive number of services in the decentralised environment is illustrated in Figure 3.10.

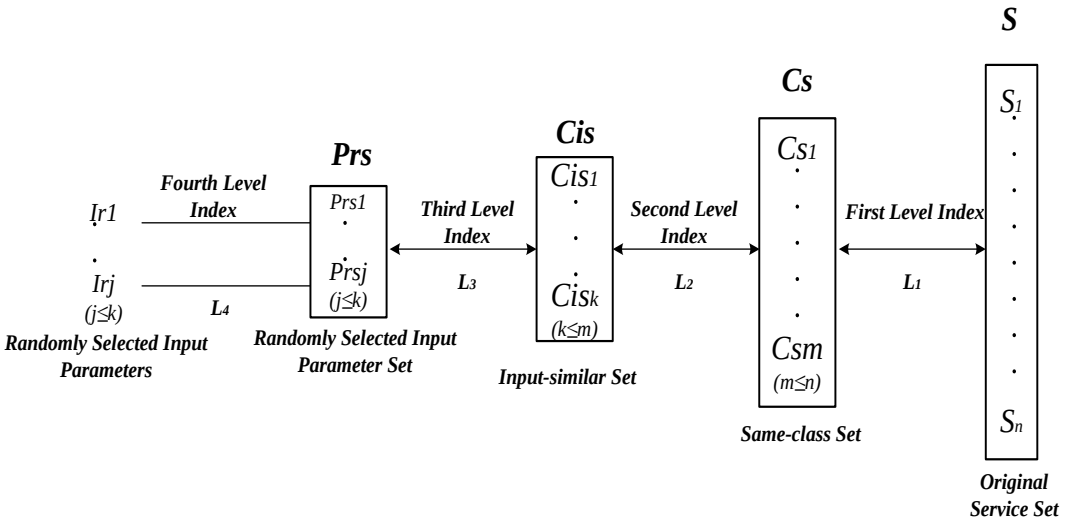

Figure 3.10 DM-Index Model

3.3.6 DM-index Operation Algorithms

As discussed in the previous section, for a given bijection function $f: X \to Y$, its invertible function f^{-1} can be denoted as $Y \to X$. Based on *Definition 3.13,* if $f: X \to Y$ is a surjection and $Z \subseteq Y$, the preimage of Z under function f

can be defined as $f^{-p}(Z) = \{x | x \in X \wedge f(x) = Z\}$. Moreover, based on *Definition 3.11* and the first level index, original service set S is partitioned by an equivalence relation R_1 into a quotient same-class C_s set and services ($s \in S$) are clustered into the same-class C_s charactering the same input and output parameters. According to *Definition 3.13,* there is a surjection function between S and C_s, defined as follows:

Function 3.1: *First Index Level Surjection Function*

$$f_1 : S \rightarrow C_s$$

Same-class C_s set is further partitioned into a quotient input-similar C_{is} set by equivalence relation R_2. According to this procedure, a surjection is formed and denoted as f_2.

Function 3.2: *Second Index Level Surjection Function*

$$f_2 : C_s \rightarrow C_{is}$$

Although, the establishing progress of the third level index is different from the first and second level indexes, all the elements of C_{is} set can be indexed by P_{rs}. Therefore there is a surjection function between C_{is} and P_{rs}.

Function 3.3: *Third Level Index Surjection Function*

$$f_3 : C_{is} \rightarrow P_{rs}$$

As all elements of P_{rs} set are only indexed by one input parameter I_r. The function f_4 between I_r and P_{rs} set is a surjection function defined as follows.

Function 3.4: *Fourth Level Index Bijection Function*

$$f_4 : P_{rs} \rightarrow I_r$$

Thus all the functions can be embedded into the proposed multilevel index model. Since every level index has the integrity without redundancy, as a whole unit, the proposed DM-index model can ensure the integrity and avoid redundancy.

1. **Service Discovery**

Service discovery is an essential operation of the proposed DM-index model. For a given user requests $Q(Q_p, Q_r)$, the retrieval process in service repositories with DM-index model can be shortlisted as following.

a) Step 1: Check whether Q_p can match at least one existing I_r or not. If so, go to step 2, otherwise return not found.

b) Step 2: Retrieve P_{rs} indexed by matched input parameter I_r.

c) Step 3: Retrieve input-similar C_{is} indexed by relevant P_{rs}, only those can satisfy $P_{cisi} \subseteq Q_p$ are chosen for further operation. If found, go to step 4, otherwise, return not found.

d) Step 4: Similar to Step 3, only those indexed C_s satisfying $Q_r \subseteq P_{cso}$ are chosen. If found, go to step 5, otherwise, return not found.

e) Step 5: Discover and return s indexed by C_s.

Based on service retrieval defined in *Definition 3.6*, the algorithm of service discovery in repositories with DM-index model is shown Algorithm 3.1.

Algorithm 3.1: *Service Discovery with DM-index Model*

---

1 $I_r' = Q_p \cap I_r$

2 $P_{rs}' = f_4^{-p}(I_r')$

3 $C_{is}' = f_3^{-p}(P_{rs}') \wedge P_{cisi} \subseteq Q_p$

---

4 $C_s' = f_2^{-p}(C_{is}') \wedge Q_r \subseteq P_{cso}$

5 $s = D\left(Q_p, Q_r, L, f_1^{-p}(C_s')\right)$

2. **Service Addition**

When a service s is inserted into a repository with DM-index in decentralised environments, the first step is to identify whether the service s is existing or not. If there is no matching, the new service s should be added to existing service set S. Then it is added to same-class C_s set, input-similar C_{is} set , P_{rs} sets and I_r list in a step by step fashion. It there is a matching, then check P_{si} and $I_{rs}$with existing P_{cisi} and I_r for further operations. The service addition algorithm is illustrated as follows in Algorithm 3.2.

Algorithm 3.2: *Service Addition with DM-index Model*

1 I_{rs} *is the randomly selected input parameter for service* s *hashing*
2 *if* $s \in S$
3 *service already existing*
4 *else if* $s \notin S$ && $P_{si} \in P_{cisi}$ && $I_{rs} \in I_r$
5 {
6 $S = S \cup s$
7 $C_s = C_s \cup s$
8 }
9 *else if* $s \notin S$ && $P_{si} \notin P_{cisi}$ && $I_{rs} \in I_r$
10 {
11 $S = S \cup s$
12 $C_s = C_s \cup s$
13 $C_{is} = C_{is} \cup P_{si}$
14 }
15 *else if* $s \notin S$ && $P_{si} \notin P_{cisi}$ && $I_{rs} \notin I_r$
16 {

17	$S = S \cup s$
18	$C_s = C_s \cup s$
19	$C_{is} = C_{is} \cup P_{si}$
20	$P_{rs} = P_{rs} \cup I_{rs}$
21	$I_r = I_r \cup I_{rs}$
22	}

3. **Service Deletion**

Service Deletion is used to delete a service s from service repository with the DM-index model. The deletion algorithm is shown in Algorithm 3.3.

Algorithm 3.3: *Service Deletion with DM-index Model*

1	$S = S - s$
2	*if* $\|S\| == 0$
3	*delete all* C_s *set,* C_{is} *set,* P_{rs} *set and* I_r *list*
4	*else* $C_s = f_1(S)$
5	$C_{is} = f_2(C_s)$
6	*if* $\exists P_{rsi} \in P_{rs} \quad f_3^{-p}(P_{rsi}) \cap C_{is} == \emptyset$
7	{
8	$I_r = I_r - f_4(P_{rsi})$
9	$P_{rs} = P_{rs} - P_{rsi}$
10	}

3.3.7 Adaptive Deployment

1. **Partial Deployment Model**

As mentioned in the previous sections, the first level index (L_1) of the proposed DM-index model aims to remove redundancy induced by services with the same input and output parameters. If a service repository does not contain

services with the same input and output parameters, the first level index yields no effect. Let, E_1 denotes the efficiency improved and L_1 denote the overhead induced by the first level index. If there are small number of services with the same input and output parameters, it is possible that $E_1 \leq L_1$. When many services share the same input and output parameters, $L_1 \leq E_1$, a threshold is needed, which may vary depending on a particular environment. In some cases, the first level index is not needed which means that services directly link to input-similar classes (C_{is}) according to their input parameters as shown in Figure 3.11, which is called as partial deployment in this research study.

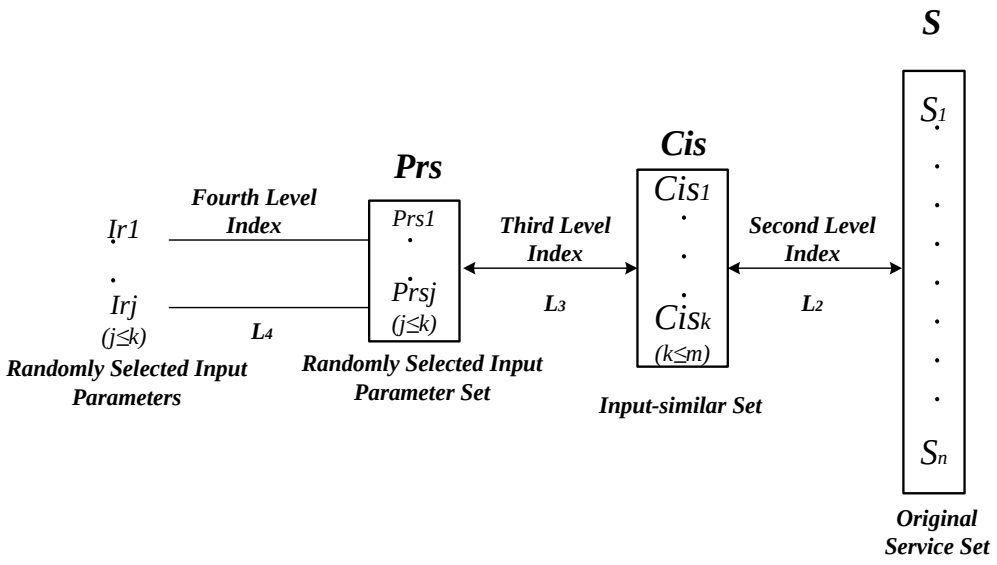

Figure 3.11 Partial Deployment Model

Service discovery algorithm of $R(Q_p, S)$ with partial DM-index deployment model is shown in Algorithm 3.4.

Algorithm 3.4: *Service Discovery with Partial Deployment Model*

1 $I'_r = Q_p \cap I_r$

2 $P'_{rs} = f_4^{-p}(I'_r)$

3 $C'_{is} = f_3^{-p}(P'_{rs}) \wedge P_{cisi} \subseteq Q_p$

4 $s = D\left(Q_p, Q_r, L, f_2^{-p}(C'_{is})\right)$

The service addition algorithm with the partial deployment model is illustrated in Algorithm 3.5.

Algorithm 3.5: *Service Addition with Partial Deployment Model*

```
1   I_rs is the randomly selected input parameter for service s hashing
2   if s ∈ S
3      service already existing
4   else if s ∉ S && P_si ∈ P_cisi && I_rs ∈ I_r
5           S = S ∪ s
6       else if s ∉ S && P_si ∉ P_cisi && I_rs ∈ I_r
7               {
8                 S = S ∪ s
9                 C_is = C_is ∪ P_si
10                }
11            else if s ∉ S && P_si ∉ P_cisi && I_rs ∉ I_r
12                {
13                  S = S ∪ s
14                  C_is = C_is ∪ P_si
15                  P_rs = P_rs ∪ I_rs
16                  I_r = I_r ∪ I_rs
17                }
```

The service deletion algorithm with the partial deployment model is illustrated in Algorithm 3.6.

Algorithm 3.6: *Service Deletion with Partial Deployment Model*

```
1   S = S − s
2   if |S| == 0
```

3	*delete all* C_{is} *set,* P_{rs} *set and* I_r *list*
4	*else* $C_{is} = f_2(S)$
5	*if* $\exists P_{rsi} \in P_{rs} \quad f_3^{-p}(P_{rsi}) \cap C_{is} == \emptyset$
6	{
7	$I_r = I_r - f_4(P_{rsi})$
8	$P_{rs} = P_{rs} - P_{rsi}$
9	}

2. **Primary Deployment Model**

Similar to the first level index, the second level index (L_2) aims to remove redundancy induced by services with the same input parameters. If a service set does not contain services with the same input parameters, the second level index yields no effect. Let, E_2 denote the efficiency improved and L_2 denote the overhead induced by the second level index. Similarly, for $E_2 \leq L_2$, the second level index is not needed. Services directly link to key classes according to their input parameters as shown in Figure 3.12, which is named as primary deployment in this research study.

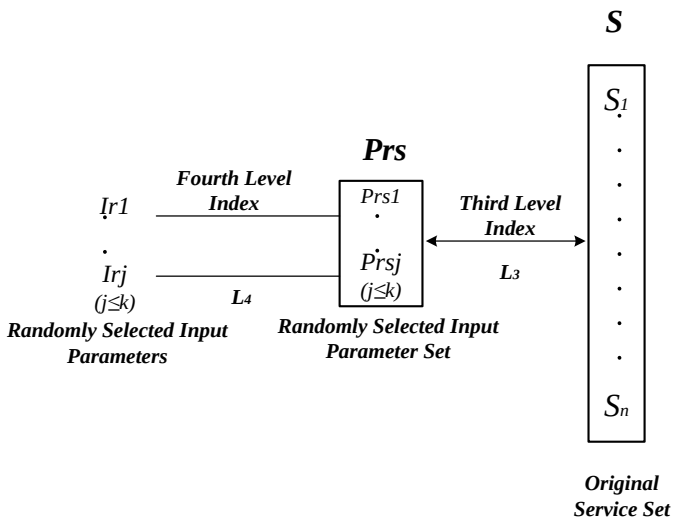

Figure 3.12 Primary Deployment Model

Similarly, service discovery algorithm of $R(Q_p, S)$ with primary deployment model is shown in Algorithm 3.7.

Algorithm 3.7: *Service Discovery with Primary Deployment Model*

1 $I'_r = Q_p \cap I_r$
2 $P'_{rs} = f_4^{-p}(I'_r)$
3 $s = D\left(Q_p, Q_r, L, f_3^{-p}(P'_{rs})\right)$

When a new service s is inserted into repository with the primary deployment model, the service addition algorithm used is illustrated in Algorithm 3.8.

Algorithm 3.8: *Service Addition with Primary Deployment Model*

1 *I_{rs} is the randomly selected input parameter for service s hashing*
2 *if* $s \in S$
3 *service already existing*
4 *else if* $s \notin S$ && $I_{rs} \in I_r$
5 $S = S \cup s$
6 *else if* $s \notin S$ && $I_{rs} \notin I_r$
7 {
8 $S = S \cup s$
9 $I_r = I_r \cup I_{rs}$
10 $P_{rs} = P_{rs} \cup I_{rs}$
11 }

To delete a service s with randomly selected input parameter I_{rs} with primary deployment model, the service deletion algorithm is presented in Algorithm 3.9.

Algorithm 3.9: *Service Deletion with Primary Deployment Model*

1 $S = S - s$

2 *if* $|S| == 0$

3 *delete both* I_r *list and* P_{rs} *set*

4 *else if* $\exists P_{rsi} \in P_{rs} \&\& \ f_3^{-p}(P_{rsi}) \notin S$

5 *{*

6 $I_r = I_r - f_4(P_{rsi})$

7 $P_{rs} = P_{rs} - P_{rsi}$

8 *}*

3. Deployment Model Selection

For a given service repository in the decentralised system, if there are no many services sharing the same input parameters, the partial model may be slightly slower than the primary model due to the complicated structures of the partial model. For similar reasons, the efficiency of the full DM-index may be slightly lower than that of the partial model. The process of selecting the best suitable deployment model for service repository is explained as follows.

a) Step 1: Deploy full DM-index model

b) Step 2: If $|S| \approx |C_s| \gg |C_{is}|$, deploy partial model,

c) Step 3: If $|S| \approx |C_s| \approx |C_{is}|$, deploy primary model.

The algorithm for adaptive deployment model selection is shown Algorithm 3.10.

Algorithm 3.10: *Adaptive Deployment*

1 *Deploy full DM-index model*

2 *If* $|S| \approx |C_s| \gg |C_{is}|$ *choose partial DM-index model*

3 *If* $|S| \approx |C_s| \approx |C_{is}|$ *choose primary DM-index model*

3.4 Theoretical Evaluation

For a retrieval request $R(Q_p, S)$, usually several services will be retrieved or traversed to determine whether they can satisfy the retrieval requirements. It can be argued that the retrieval time is a more accurate and objective value than the traversed service count. The works of [31] proposed a sampling test method that uses retrieval time as a criterion to make a decision. It is an indirect method and depends on sample sets. But this thesis aims to propose a direct, quicker and more straightforward method to choose the optimum indexing model.

For different retrieval requests, the traversed service counts may be different with different index models. Their traversed service counts vary around a theoretical average value, called as excepted value in this research study. Obviously, for an efficient indexing model, this expected value should be as low as possible. Therefore, in this research study, the expected value of the traversed service count is adopted as a criterion to determine the optimal structure of the proposed DM-index for a given service repository.

Different indexing models have different structures. Therefore, the expected values of the traversed service count of various indexing models are different. Furthermore, different service sets have different characteristics such as service count and parameter count. According to these features and the proposed formulations of the expected values, the optimal index can be decided. In this section, sequential index, inverted index, primary deployment model, partial deployment model and the full DM-index models are analysed and compared. Their functions of the expected value of the traversed service count for a retrieval request are proposed based on the assumption that invoked frequencies of all services are equal.

Definition 3-15: *The number of input parameters pool*

$$P = P_{sji} \cup P_{ski} \quad (\forall s_j, s_k \in S)$$

P represents the input parameters pool for all stored services.

Definition 3-16: *The average number of input parameters of service*

$$P_i = \overline{|P_{sjl}|} \qquad (\forall s_j \in S)$$

$P_i(NPI)$ denotes the average input parameters count of each stored service.

Definition 3-17: *The average number of provided parameters of retrieval request*

$$P_r = \overline{|Q_p|}$$

$P_r(NPR)$ represents the average provided parameters count of each retrieval request $Q(Q_p, Q_r)$.

3.4.1 Sequential Index Model

The sequential index model is also known as a non-index model, in which all the services are stored in a sequential structure, such as list. For a given retrieval request, all the services will be traversed and checked whether they match the request or not. The expectation of the traversed services for sequential index (E_s) is easy to be defined as in *Equation 3.1 and* is equal to the number of stored services.

Equation 3.1

$$E_s = |S| \qquad (1)$$

where E_s denotes the expectation of the sequential index and |S| denotes the number of stored services in a repository. For example, Figure 3.13 presents a

service repository containing five services, the excepted value E_s for the sequential index is 5 since |S|=5.

$$s_1{:}ab{\rightarrow}cd \quad s_2{:}ab{\rightarrow}cd \quad s_3{:}ad{\rightarrow}ce \quad s_4{:}ad{\rightarrow}ef \quad s_5{:}cd{\rightarrow}gh$$

Figure 3.13 Example of Service Retrieval with Sequential Index

3.4.2 Inverted Index Model

Inverted index model is one of the most popular indexes which can narrow the search space and improve the retrieval efficiency. Figure 3.14 shows an example of service retrieval with the inverted index.

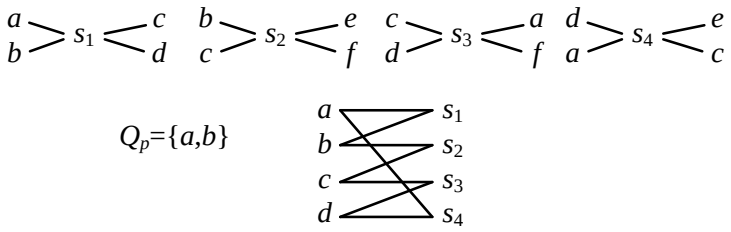

Figure 3.14 Example of Service Retrieval with Inverted Index

In this example, there are a couple of index links between the input parameters $P_{si}\,(a,\ b, c\ , d)$ and the services s_1, s_2, s_3 and s_4. Given a retrieval request $Q_p = \{a, b\}$, both services s_1 and s_4 will be traversed by parameter a and both services s_1 and s_2 will be traversed by b,. Finally, s_1 will be returned. From the example, it can be concluded that only part of elements in service set S is traversed. As mentioned in Chapter 2, only a part of the service set is traversed by deploying the inverted index model. However, service s_1 is accessed twice. Let E_i denote the expectation of the traversed services for inverted index, the expectation of the inverted index (E_i) can be defined as in *Equation 3.2*.

Equation 3.2

$$E_i = \frac{P_i \times P_r}{|P|} \times |S| \quad (2)$$

where E_i denotes the expectation of the inverted index and all services have $P_i \times |S|$ index items. Services are scattered on $P_i \times |S|/P$ means that the average number of services are linked by one input parameter. Then $P_i \times P_r \times |S|/P$ denotes the number of services those will be traversed for a given request$Q(Q_p, Q_r)$. Clearly, if $P_i \times P_r > |P|$, its efficiency is lower than the sequential index. To the given example, $|S|$ *=4,* $|P|$ *=4,* $|p_{si}|$ *=2* and E_i*=2*$|Q_p| = 4$.

3.4.3 DM- Index Model

1. **Full Level Model**

For the MD-index full deployment model, the service set S of a repository is divided into many subclasses C_s based on the equivalence relation R_1. Furthermore, the same-class C_s set can be divided into many input-similar subclasses C_{is} based on the equivalence relation R_2. Moreover, only one indexed input similar subclass satisfies $C_{isj} \in C_{is} \wedge P_{cisi} \subseteq Q_p$ and one indexed same-class satisfies $C_{si} \in C_s \wedge Q_r \subseteq P_{cso}$ will be retrieved step by step according to *Algorithm 3.1*. Therefore the expectation E_{fl} of the full level DM-index model can be presented as in *Equation 3.3.*

Equation 3.3

$$E_{fl} = |Q_p \cap I_r| \times \frac{\overline{|C_s|}}{|C_{is}|} \quad (3)$$

where $\frac{\overline{|C_s|}}{|C_{is}|}$ represents the average count of clustered services by each input-similar subclass $\forall C_{isj} \in C_{is}$. According to prior analysis, the size of same-class C_{si} set should no more than original set ($\frac{\overline{|C_s|}}{|C_{is}|} \leq \frac{\overline{|S|}}{|C_{is}|}$) in full deployment model. $|Q_p \cap I_r|$ means that the number of matched input parameters I_r between a given users requirement Q_p and the existing fourth index parameters list. All the parameters of $Q_p$will be retrieved in the worst case.

2. Primary Model

As described in the previous section, the full level DM-index model can be used instead of the primary deployment model when a service set contains only a fewer services with the same input parameters. For the primary deployment model, there are no first and second level indexes. The expectation is denoted by E_{pr} and can be calculated by the following formula.

Equation 3.4

$$E_{pr} = |Q_p \cap I_r| \times \frac{\overline{|S|}}{|I_r|} \qquad (4)$$

As there are no same-class C_s set or input-similar class C_{is} set in the primary deployment model, the expectation of primary deployment model E_{pr} only relates to the number of randomly selected input parameters I_r. The average number of services indexed by each input parameter is donated as $\frac{\overline{|S|}}{|I_r|}$. Based on *Function 3.4*, f_4 is a bijection function between I_r set and P_{rs} set. Thus the number of randomly selected input parameters in I_r list is equal to the size of P_{rs} set in the third level. For a given request $Q(Q_p, Q_r)$, $\left|Q_p \cap I_r\right| = \left|Q_p \cap P_{rs}\right| \leq |I_r|$ even in the worst circumstance. Therefore, the efficiency of the primary index is not lesser than the sequential model.

3. Partial Model

For the partial deployment model, the expectation can be represented by E_{pt} and estimated in the following formula.

Equation 3.5

$$E_{pt} = |Q_p \cap I_r| \times \overline{\frac{|S|}{|C_{is}|}} \quad (5)$$

As there are no same-class C_s set in the partial model, the average number of services maintained by each input-similar subclass is $\overline{\frac{|S|}{|C_{is}|}}$. The efficiency of the partial index may range between the primary index E_{pr} and the full level E_{fl} since $|C_{is}| \geq |I_r|$ and $|S| \geq |C_s|$.

In simple words, the service discovery efficiency increases from low to high for the sequential index, inverted index, primary model of DM-index, partial model and full level DM-index respectively.

3.5 Summary

As the first main contribution of this thesis, the proposed DM-index model is an efficient approach for service maintenance, retrieval and discovery. The proposed model can effectively eliminate the redundancy information, which is usually incurred by the same and input-similar services. Theoretical evaluations demonstrate that the proposed DM-index model can improve the service retrieval efficiency than the sequential index and inverted index models.

4 Double-layer No-redundancy Enhanced Bi-direction Chord

A wide range of research works have focused on enhancing the service discovery efficiency in decentralised systems, which includes a number of Chord-based protocols to exploit proximity, reduce the number of traversed nodes for quicker query resolution. However, most of these solutions may lose efficiency in a decentralised system comprising larger number of distributed service repositories with massive stored services. This chapter presents an efficient searching algorithm with double-layer routing mechanism and optimised routing index in order to achieve efficient service discovery in decentralised environments.

4.1 Decentralised Environment with Chord

As discussed in the literature review, Distributed Hash Tables (DHTs) is an efficient solution that can provide *put (key, value) / get (key)* interfaces to store and retrieve information in a structured distributed network. Clients and routers store the *(key, value)* pairs in which data items, services or the location are usually mapped on to the keys [94]. In general, the key of a given resource is a hash of that particular resource name, achieved through a hashing function defined by the DHTs algorithms, while the value is a short data corresponding to that resource [151]. DHTs rely on the cooperation of certain nodes which collectively provide the information storage and retrieval services. Nodes of DHTs are arranged on an overlay network, which is built upon an existing network, whose topology is decided by the nature of the DHTs algorithms.

In Chord, each repository (R) and service (s) is assigned with a m-bit ID using a base hash function such as SHA-1. Repositorys' IDs are ordered in an

ID circle modulo 2^m. The basic principle is to store the service in the first repository whose ID is equal to or follows the service's ID [33]. Figure 4.1 presents an example of the distributed repositories of decentralised system organised in Chord circle with $m = 6$, in this structure the Chord circle is capable of storing $2^6 = 64$ different repositories and services. Assuming that five services will be stored in fifteen repositories, the successor of service with identifier *S9* is repository *R10*, so *S9* is located at *R10*. Similarly, *S18* is stored at *R20*, *S30* is contained in *R31*, *S43* is stored in *R44* and *S55* is saved in repository *R57* respectively.

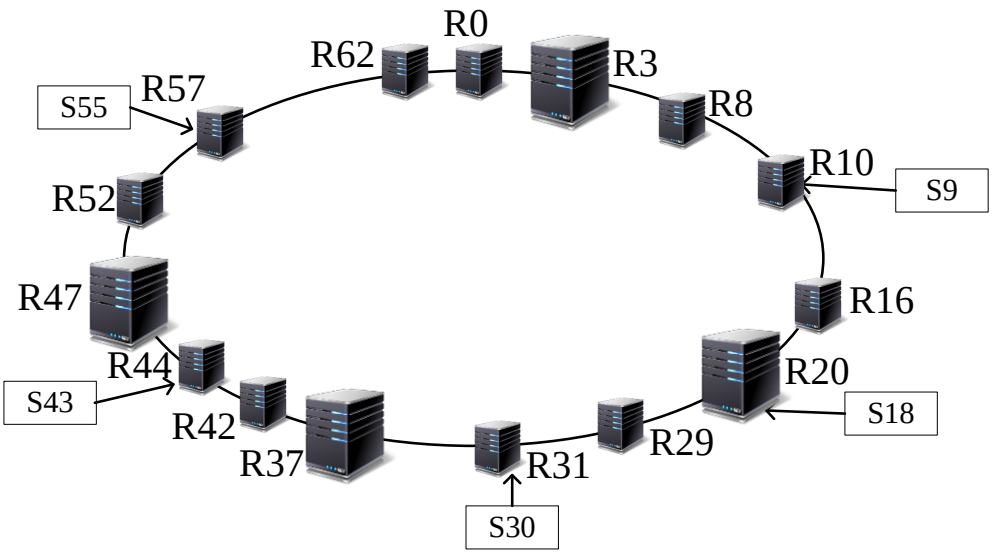

Figure 4.1 Decentralised System Organisation Based on a Chord Circle

In a network consisting of N repositories, when all the neighbours are maintained (the full directory case) by these repositories, the search cost is $O(1)$. When each repository only maintains one neighbour (successor), the search cost is $O(N)$ [110]. In practical systems, neither extreme is desirable because of the heavy maintenance burden or routing cost. Instead each repository maintains a routing index consists of $i(1 \leq i \leq m)$ entries [152], when the length of the service/repository hash identifiers is m. The ith entry

in the repository *R's* routing index represents the first succeed repository on circle, i.e. $R*$ = successor $(R + 2^{k-1}) mod 2^m$, where $1 \leq k \leq m$ [33]. The typical definition of routing index is illustrated in Table 4.1 [1].

Table 4.1 Typical Routing Index

Notation	Definition
index[k]	First repository on circle succeed $(R + 2^{k-1}) mod 2^m, 1 \leq k \leq m$
index[k].ID	Identifier of the $k-th$ repository
index[k].Address	Address of the $k-th$ repository
successor	The next repository on the identifier circle

For the same example listed in Figure 4.1, assuming there are five entries in repository *R3's* routing index, the first entry in repository *R3*'s routing index is *R8* because $(3 + 2^{1-1}) mod 2^6 = 4$. Similarly, both the second and third entries are *R8* and the fourth entry is *R16.* Now, the routing index with five entries for repository *R3* is demonstrated in Figure 4.2.

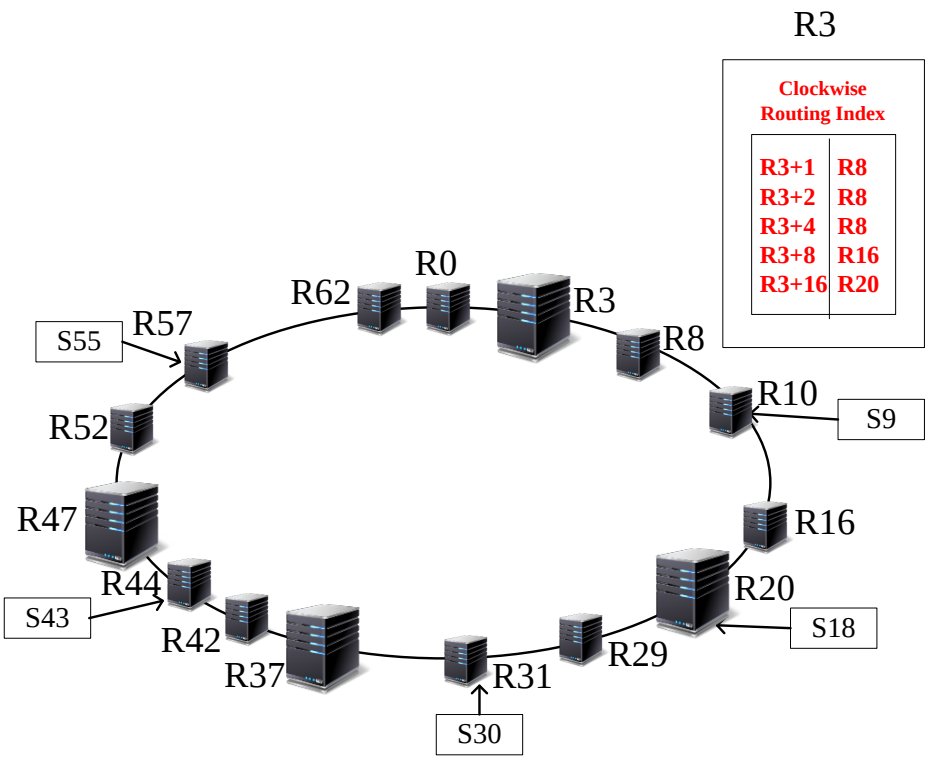

Figure 4.2 R3 with Traditional Routing Index

As searching and locating are essential operations, the basic searching operation for repository *R* to find out service *s* based on Chord is explained as follows [104]:

a) Step 1: Searching in repository *R* itself firstly, if the searching is successful, return *s*. Otherwise, go to step 2.

b) Step 2: Searching in repository *R's* routing index. If service *s* is stored in a one-hop neighbour repository *R**, return *s*. Otherwise, go to step 3.

c) Step 3: The query is forwarded and stored to new repository *R** which is the closest predecessor of *s* in *R's* routing index, the current repository *R* is replaced by *R** and repeat step 2 until the searching process is successful.

This searching operation based on Chord is presented in *Algorithm 4.1*.

Algorithm 4.1: *Service Discovery Based on Chord*

```
1  R.s_find(s)
2  if  s  stored in R
3       return  s
4  else if  s  stored in a one-hop neighbour repository R*
5            return  s
6       else
7            R*=the closest predecessor repository of s in R's routing index
8            return R*.s_find(s)
```

Now for the same example presented in Figure 4.2, how can repository *R3* discover service *S43*? Since *R20* is the closest preceded repository of *S43* among *R3's* routing index, the query will be forwarded to *R20* firstly. Then, the closest predecessor *R37* is selected for *S43* based on *R20's* routing index. Thirdly, the query is forwarded to *R42* by *R37*. Finally, *R42* finds that *S43* is stored in its one-hop neighbour *R44*. Thus, the final service *S43* is returned to *R3* from the destination repository *R44*. The red arrow in Figure 4.3 illustrates the path of the entire service discovery operation based on Chord.

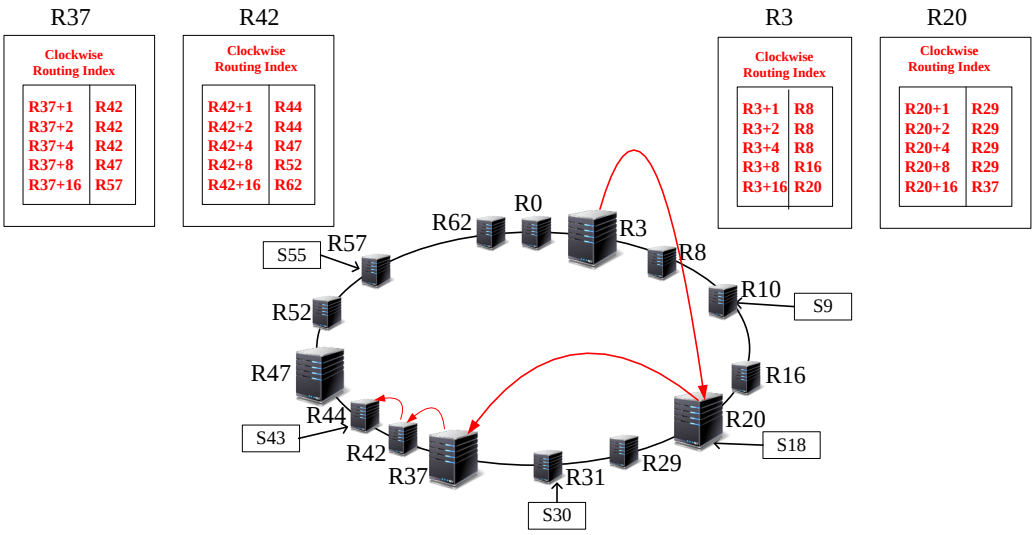

Figure 4.3 Service Discovery Based on Chord

Assuming that the prior proposed DM-index model has been successfully implemented in each repository, service discovery should consist of two parts in this circumstance including locating the destination repository and retrieving services from the destination repository with the DM-index model. The service discovery algorithm is explained in *Algorithm 4.2* by combining *Algorithm3.1* and *Algorithm 4.1*.

***Algorithm 4.2**: Service Discovery with DM-index and Chord*

---

1 *// locating the destination repository with requested service hash* ID_s

2 *R.find(*ID_s*)*

3 *if* ID_s *stored in R*

4 *return R*

5 *else if* ID_s *stored in a one-hop repository R**

6 *return R**

7 *else*

8 *R*=the closest repository* $(< ID_s)$ *in R's routing index*

9 *return R*.find(*ID_s*)*

---

10 *// Discovering service from destination repository R with DM-index*

11 $I'_r = Q_p \cap I_r$

12 $P'_{rs} = f_4^{-p}(I'_r)$

13 $C'_{is} = f_3^{-p}(P'_{rs}) \wedge P_{cisi} \subseteq Q_p$

14 $C'_s = f_2^{-p}(C'_{is}) \wedge Q_r \subseteq P_{cso}$

15 $s = D\left(Q_p, Q_r, L, f_1^{-p}(C'_s)\right)$

For the same example discussed in Figure 4.1, Figure 4.4 shows the process of discovering service *S43* from *R3* in the DM-index model implementation based on the *Algorithm 4.2*. Firstly, the destination repository *R44* can be found in four hops from *R3* based on Chord. Then the required service $s = \{s | s \in f_1^{-p}(C'_s) \in S \wedge P_{si} \subseteq Q_p \wedge Q_r \subseteq P_{so} \wedge s \angle L\}$ can be quickly retrieved from repository *R44* based on the DM-index model.

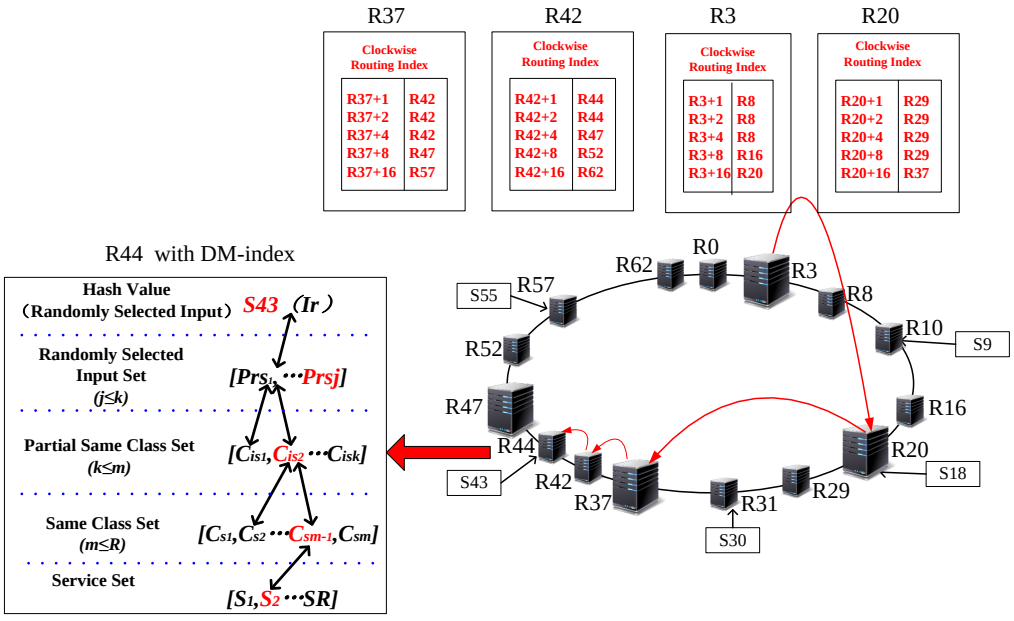

Figure 4.4 Service Discovery Based on DM-index and Chord

4.2 No-redundancy Enhanced Routing Index

The routing index based on Chord only contains the information in clockwise direction; this is the reason why it is usually inefficient for achieving efficient service discovery and retrieval among all the repositories [104]. With this in mind, this section is to propose an efficient searching algorithm which can optimise the routing mechanism and routing index with less maintenance burden, lower routing cost and fewer traversed repositories.

4.2.1 Bi-direction Routing Index

From Figure 4.3, it can be observed that the query can only be resolved in clockwise direction since the routing index does not contain enough information for efficiently locating the counter-clockwise objective services. Therefore, a counter-clockwise routing index will be introduced to each repository in the first stage to propose the upcoming efficient searching algorithm in this chapter. Then every repository in the decentralised system should maintain two routing indexes. One holds clockwise neighbours information and another holds counter-clockwise neighbours information. A bi-directional searching mechanism and scheme is proposed as an extension, to improve the searching efficiency by merely cloning the clockwise index in the counter-clockwise direction [153]. To achieve this, the principal requirement is to establish a new counter-clockwise routing index for repository R as shown in Table 4.2.

Table 4.2 Counter-clockwise Routing Index

Notation	Definition
index[k]	The first repository on counter-clockwise circle preceded $(R - 2^{k-1} + 2^m) mod 2^m, 1 \leq k \leq m$
index [k].ID	Identifier of the $K-th$ repository
index [k].Address	Address (IP) of the $K-th$ repository
predecessor	The previous repository on the identifier circle

For the same repository discussed in Figure 4.1, *R3* maintains one more routing index with blue entries which holds the information of backward repositories as demonstrated in Figure 4.5.

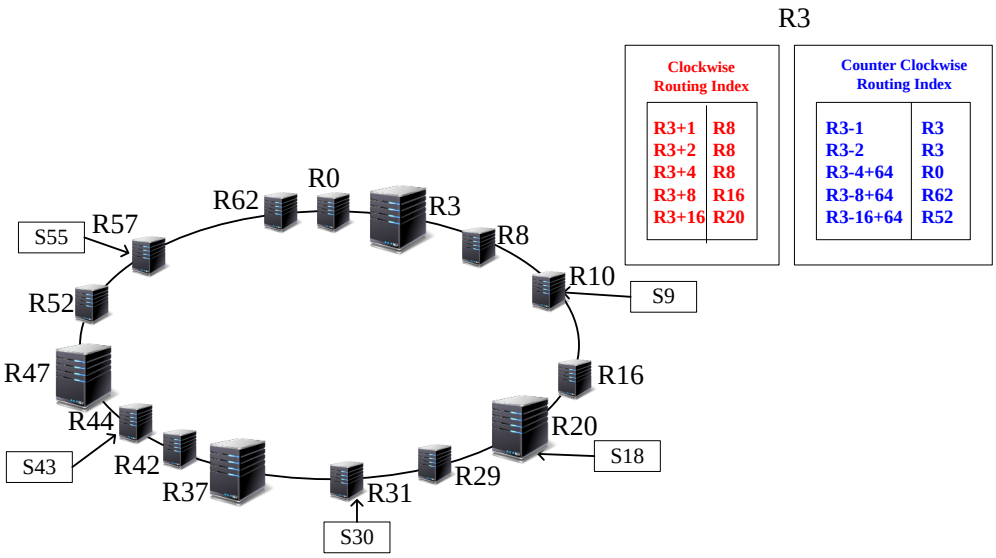

Figure 4.5 R3 with Bi-direction Routing Index

Based on bi-direction routing index, the searching operation for repository R to find service s explained as follows:

a) Step 1: Searching in R itself firstly, if the searching is successful, return s. Otherwise, go to step 2.

b) Step 2: Searching in repository *R's* bi-direction routing index. If *s* is stored in a one-hop neighbour repository *R**, return *s*. Otherwise, go to step 3.

c) Step 3: The query is transferred to the closest neighbour repository *R** in *R's* bi-direction routing index, current repository *R* is replaced by *R** and go back to step 2.

Similar to *Algorithm 4.1*, *Algorithm 4.3* presents the service discovery algorithm for repository *R* with bi-direction routing index.

Algorithm 4.3: *Service Discovery with Bi-direction Routing Index*

```
1   R.s_find(s)
2   if  s stored in R
3       return  s
4   else if  s  stored in a one-hop neighbour repository R*
5               return  s
6       else
7               {
8               R*=the closest neighbour repository in R's bi-direction
    routing index
9               return R*.s_find(s)
10              }
```

For the same case discussed in Figure 4.3, Figure 4.6 shows different searching progress. With the bi-direction routing index, *R3* checks both two routing indexes before transferring the query. Similar to the clockwise routing index, the lookup message is transmitted to the closest repository *R52* from *R3* based on the counter-clockwise routing index. Then, this lookup message will arrive at the destination *R44* which stores *S43*. From the blue arrows represented in

Figure 4.6, it can be observed that it only takes two hops to get *S43* with the bi-direction routing index rather than four hops with the clockwise routing index only.

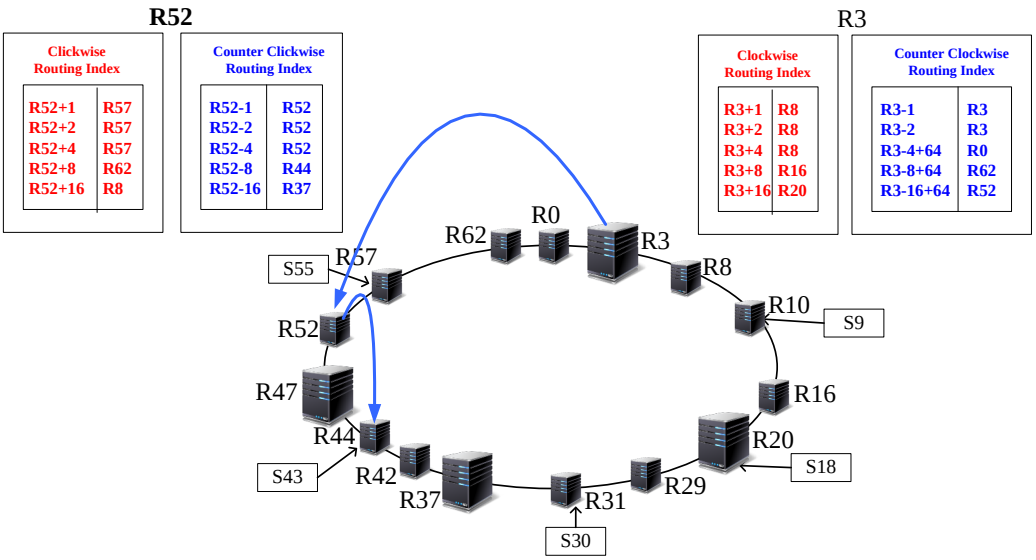

Figure 4.6 Searching Paths with Bi-direction Routing Index

By implanting the DM-index and bi-direction routing index to each repository in decentralised systems, service discovery algorithm is listed in *Algorithm 4.4* by integrating *Algorithm 3.1* and *Algorithm 4.2*. It consists of two parts: finding the destination repository in the decentralised environment and discovering the required services using the DM-index model.

Algorithm 4.4: *Service Discovery with Bi-direction Routing Index and DM-index*

---

1 *// locating the destination repository with requested service hash* ID_s

2 *R.find(*ID_s*)*

3 *if* ID_s *stored in R*

---

4 *return R*

5 *else if* ID_s *stored in a one-hop neighbour repository R**

6 *return R**

7 *else*

8 *{*

9 *R*=the closest repository in R's bi-direction routing index*

10 *return R*.find(*ID_s*)*

11 *}*

12 *//Discovering service* s *in the destination repository R with DM-index*

13 $I'_r = Q_p \cap I_r$

14 $P'_{rs} = f_4^{-p}(I'_r)$

15 $C'_{is} = f_3^{-p}(P'_{rs}) \wedge P_{cisi} \subseteq Q_p$

16 $C'_s = f_2^{-p}(C'_{is}) \wedge Q_r \subseteq P_{cso}$

17 $s = D\left(Q_p, Q_r, L, f_1^{-p}(C'_s)\right)$

Figure 4.7 illustrates the process to discover service *S43* from *R3* in a decentralised environment with the DM-index model and the bi-direction routing index. Firstly, the destination repository *R44* can be found within fewer hop counts by deploying the bi-direction routing index. Secondly, the service $\mathrm{s} = \left\{\mathrm{s} \middle| s \in f_1^{-p}(C'_s) \in S \wedge P_{si} \subseteq Q_p \wedge Q_r \subseteq P_{so} \wedge s \angle L\right\}$ can be efficiently discovered in repository *R44* with the help of the DM-index model.

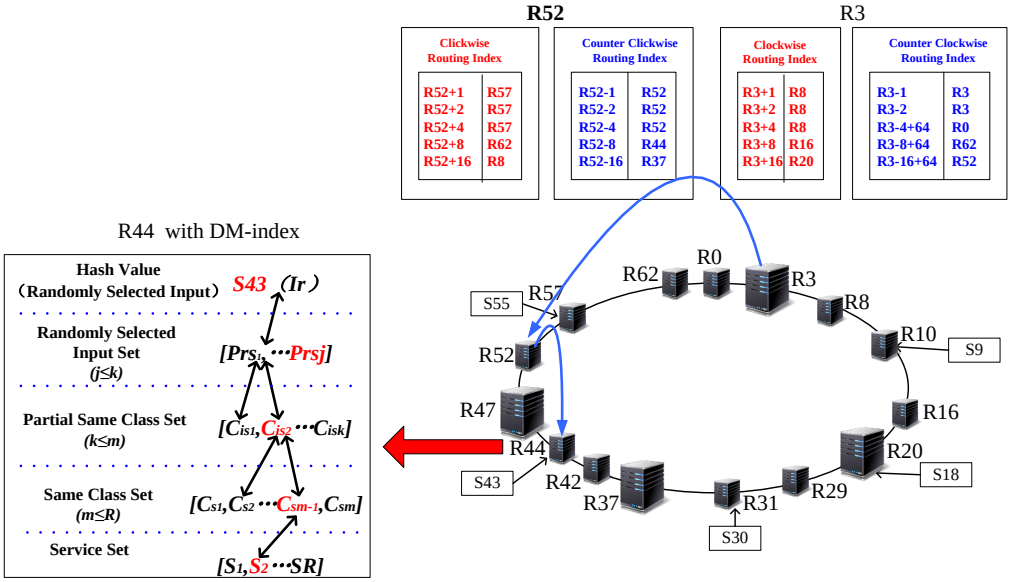

Figure 4.7 Service Discovery with DM-index and Bi-direction Routing Index

Comparing Figure 4.3 with Figure 4.6 and Figure 4.4 with Figure 4.7, it can be observed that the service discovery efficiency of decentralised systems can be improved by deploying the optimised bi-direction routing index especially when $(R - S + 2^m) mod 2^m > 2^{m-1}$.

4.2.2 Enhanced Routing Index

Although the counter-clockwise routing index may improve the search efficiency, the span of the two neighbouring repositories of both the clockwise and counter-clockwise indexes keep increasing. The span between two neighbouring repositories is strictly limited to $2^m - 2^{m-1} (m \geq 1)$. The solution to ehance the bi-direction routing index is very simple and can be achivieved by inserting new index into the middle of the two neighbouring repositories in both directions. Specifically, the bi-direction routing index of

repository R are now modified as explained in Table 4.4 and Table 4.5, where $k \leq 2m - 2$[154].

Table 4.3 Enhanced Clockwise Routing Index

Notation	Definition
index [k]	$\begin{cases} (N+1) mod 2^m & k = 1 \\ \left(N + 2^{k/2}\right) mod 2^m & k\ is\ even \\ \left(N + 2^{(k-1)/2} + 2^{(k-3)/2}\right) mod 2^m & k\ is\ odd \end{cases}$
index [k].ID	Identifier of the $K-th$ repository
index [k].Address	Address of the $K-th$ repository
successor	The next repository on the identifier circle

Table 4.4 Enhanced Counter-clockwise Routing Index

Notation	Definition
index [k]	$\begin{cases} (N-1+2^m) mod 2^m & k = 1 \\ \left(N - 2^{k/2} + 2^m\right) mod 2^m & k\ is\ even \\ \left(N - 2^{(k-1)/2} - 2^{(k-3)/2} + 2^m\right) mod 2^m & k\ is\ odd \end{cases}$
index [k].ID	Identifier of the $K-th$ repository
index [k].Address	Address of the $K-th$ repository
predecessor	The previous repository on the identifier circle

After this enhancement, repository *R3* maintains two enhanced routing indexes in both directions as shown in Figure 4.8. The yellow entries in enhanced bi-direction routing index are newly added neighbouring repositories. Note that, in comparison with Figure 4.6, new one-hop neighbour repository *R10* is added to the clockwise routing index and *R57* is added to the counter-clockwise routing index after extension [154].

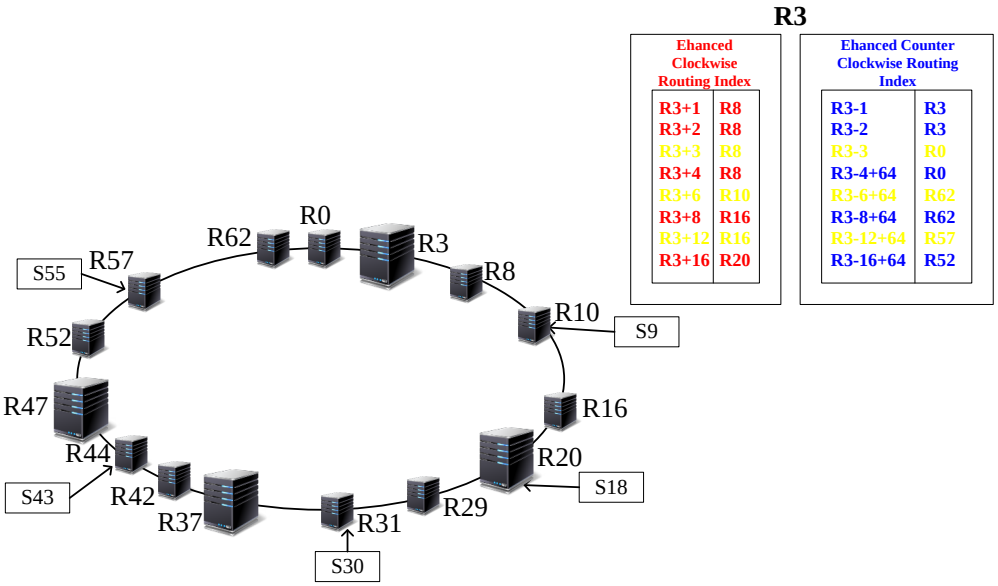

Figure 4.8 R3 with Enhanced Bi-direction Routing Index

In this case, to search the repositories holding *S9* and *S55* from *R3*, Figure 4.9 demonstrates that the destination repositories *R10* and *R57* can be allocated in one hop with the enhanced bi-direction routing index as illustrated by the solid yellow lines rather than two hops with the bi-direction routing index as illustrated by the red and blue dot lines. So this example further confirms that the enhanced bi-direction routing index can achieve better discovery efficiency in both directions with fewer traversed repositories.

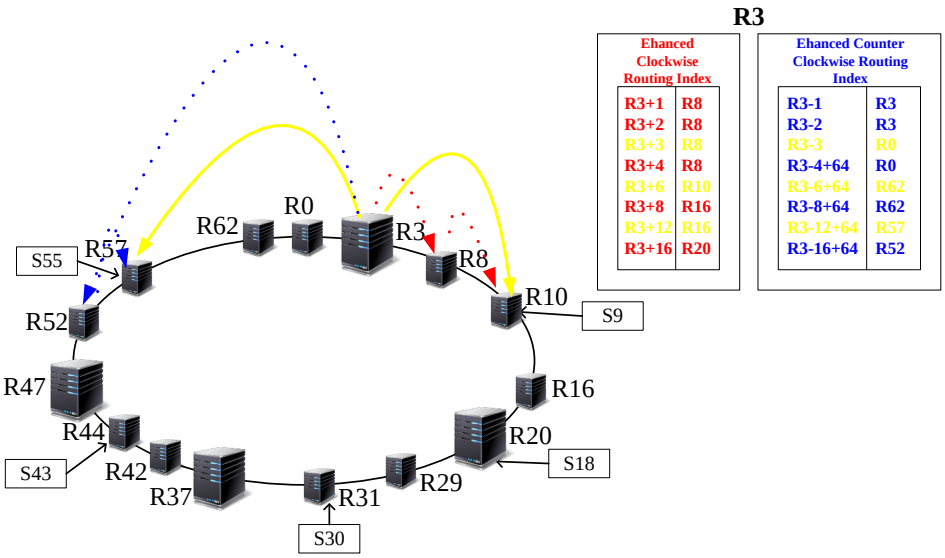

Figure 4.9 Searching Paths with Enhanced Bi-direction Routing Index

As the newly added entries do not change routing method, *Algorithm 4.4* is still suitable for repositories with enhanced bi-direction index. According to *Algorithm 4.4*, the process of discovering service *S9* and *S55* from *R3* in a decentralised environment with the DM-index model and enhanced bi-direction routing index is presented in Figure 4.10 with fewer traversed repositories. However, more entries in the enhanced bi-direction routing index bring more maintenance burden. Dealing with this extra maintenance burden to achieve better efficiency will be discussed in the following sections.

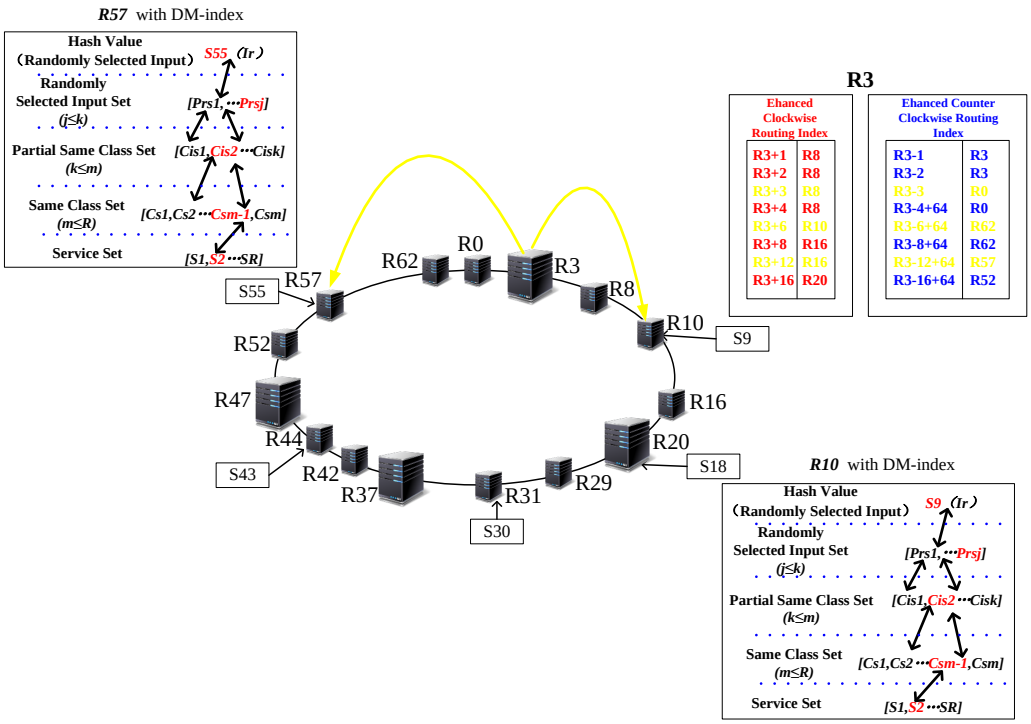

Figure 4.10 Service Discovery with DM-index and Enhanced Routing Index

4.2.3 Optimal Routing Index

In comparison with the sizes of entries included in traditional routing index (*m* entries) and bi-direction index (*2m* entries), the number of entries in enhanced bi-direction routing index is increased to *2*(2m-2)*. From Figure 4.8, it could be seen that enhanced bi-direction index may include some redundant entries those cannot support the looking up at all. These redundant entries can make the corresponding index to have different start value but the same object repository value as another index. For the example shown in Figure 4.8, there are seven redundant entries (black entries) in the enhanced bi-direction routing index as shown in Figure 4.11, which is roughly accounting for 50%.

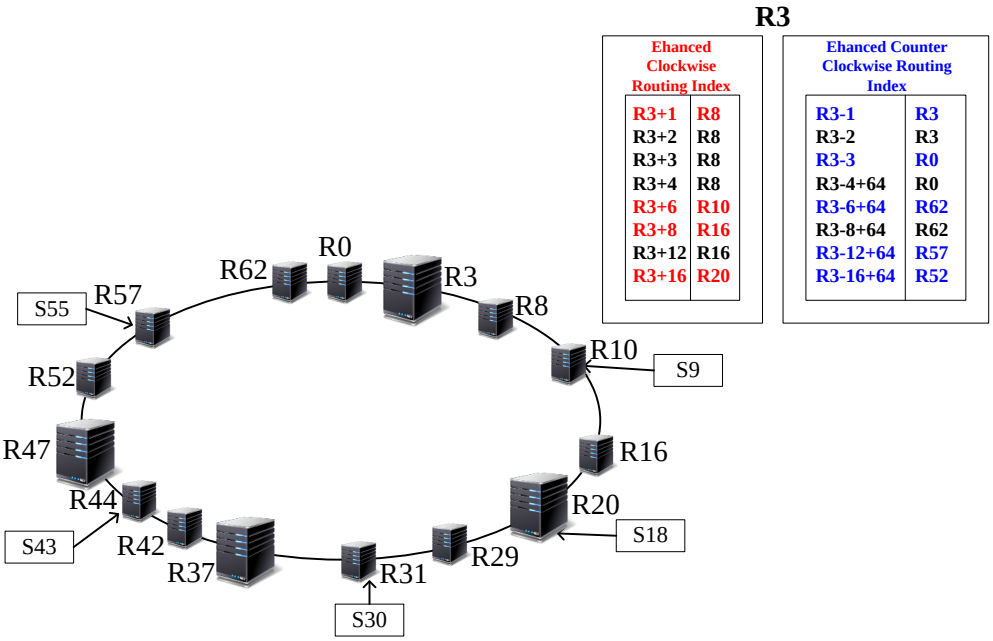

Figure 4.11 R3 with Enhance Routing Index Includes Redundancy

Considering that, in a realistic application where the number of m is as high as 160, which may lead to a lot of redundant entries. It will significantly increase the overheads of maintaining burden and updating the routing index. The redundant information in the routing index not only takes up valuable space but also increases the search delay. Therefore, it is necessary to resolve this problem. Thus, this research study tries to reduce the maintenance burden by deleting the unwanted redundant information. In detail, each repository traverses all its entries, if the redundant entries are found, then only the first index entry is maintained and the successive interval will be integrated as the first index's property of interval. For the same repository *R3*, Figure 4.12 demonstrates the repository with optimal routing index. In comparison with Figure 4.5, it can be observed that repository *R3* can reach ten neighbour repositories in one hop rather than six repositories with the same maintenance

burden. Figure 4.12 indicates that each entry can maintain one unique neighbour repository without redundant entries. In simple words, with optimal routing index, the repository can reach many more neighbouring repositories in one hop without increasing too much maintenance burden.

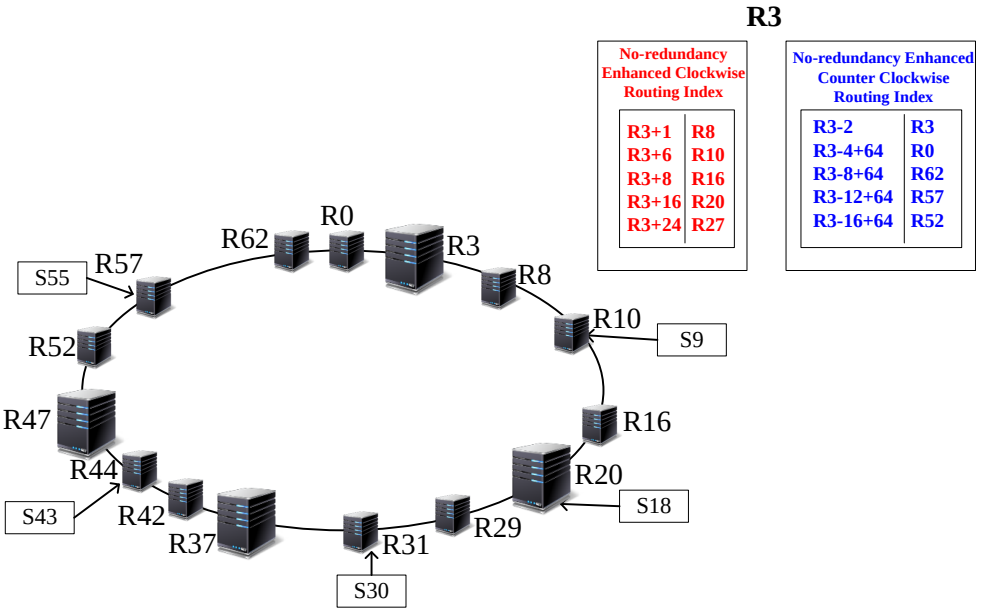

Figure 4.12 R3 with Optimal Routing Index

Whilst looking for service *S30* stored in repository *R31*, Figure 4.13 presents that the destination repository *R31* can be reached in two hops with the optimal routing index represented by the red solid arrows, rather than three hops with the enhanced bi-direction routing index represented by the black dot arrows. Therefore this example further confirms that the optimal routing index without redundant entries can achieve much better efficiency.

In conclusion, with the help of the optimal routing index, repositories can resolve the service location in less than $O(log_2 {}^{n}/_{2})$ hops [155]. This means

that each repository can forward a request of a service to its destination repository in fewer hops with less maintenance burden.

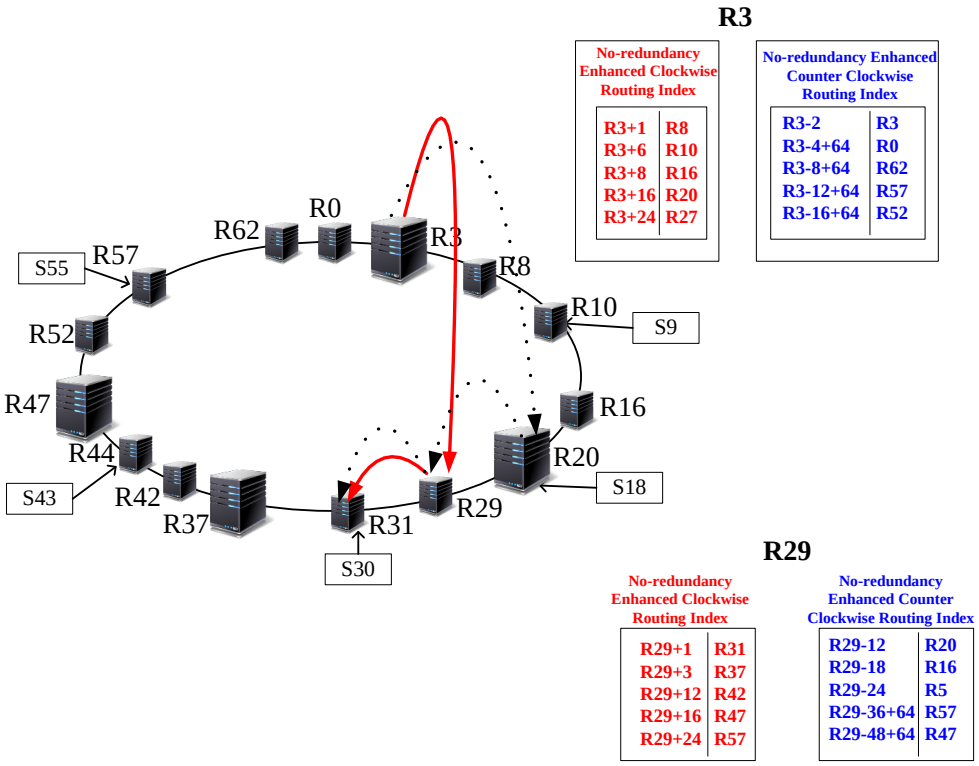

Figure 4.13 Searching Paths with Optimal Routing Index

Based on the implementations of the proposed DM-index and optimal routing index, service discovery algorithm is illustrated in *Algorithm 4.5*.

Algorithm 4.5: *Service Discovery with Optimal Routing Index and DM-index*

---

1 *// locating the destination repository with requested service hash* ID_s

2 *R.find*(ID_s)

3 *if* ID_s *stored in in R*

4 *return R*

5 *else if* ID_s *stored in a one-hop neighbour repository R**

---

6 *return R**

7 *else*

8 *{*

9 *R*=the closest neighbour repository in R's optimal routing index*

10 *return R*.find(ID_s)*

11 *}*

12 *//discovering service* s *in the destination repository R with DMindex*

13 $I'_r = Q_p \cap I_r$

14 $P'_{rs} = f_4^{-p}(I'_r)$

15 $C'_{is} = f_3^{-p}(P'_{rs}) \wedge P_{cisi} \subseteq Q_p$

16 $C'_s = f_2^{-p}(C'_{is}) \wedge Q_r \subseteq P_{cso}$

17 $s = D\left(Q_p, Q_r, L, f_1^{-p}(C'_s)\right)$

The searching process of *R3* discovers service *S30* with the DM-index model and the optimal routing index are demonstrated in Figure 4.14. Firstly, the destination node *R31* can be found within two hop counts by deploying the optimal routing index. Secondly, the service *S30* can be efficiently discovered in repository *R31* with the help of the DM-index model.

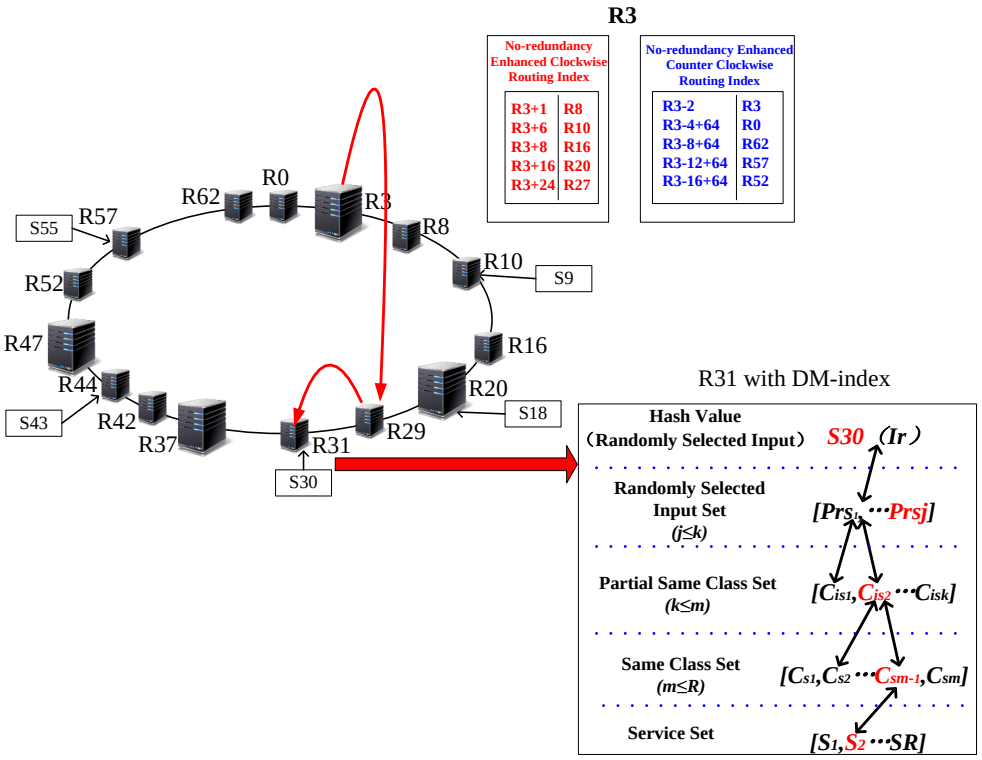

Figure 4.14 Service Discovery with DM-index and Optimal Routing Index

4.3 Double-layer Routing Mechanism

Based on DHTs, each repository plays equivalent roles in the network, which further assumes that each repository has uniform storage and bandwidth capacity. Apparently, this assumption is different from the real decentralised system, since the resources contributed by each repository in practice can be dissimilar. Therefore, the routing performance in a decentralised system with larger number of distributed repositories is usually limited by the repository with the minimum performance and resources. However, the proposed approach attempts to further improve the routing performance in overlay networks by exploiting the repository heterogeneity [156].

This research study proposes a double-layer routing mechanism with an upper-layer to which repositories can join and leave voluntarily depending on their resource availability. An upper-layer routing ring is created in the proposed approach to utilise the expressway functionality and the expressway routing performance. Moreover, any repository can join and leave the upper ring as super-repository voluntarily depending on whether they can possess more resources and bandwidth or not. Rather than merely acting as a representative of a set of normal repositories, the super-repository comprises more knowledge about the neighbours than those of the other repositories in the lower-layer. As only part of repositories could act as super-repositories, the upper-layer could forward the search requests to super-repositories in longer hop distance than the lower-layer repository. If there is no any other super-repository present along the path of the destination, search requests are delivered to the lower-layer normal routing system, which can still guarantee routing correctness. The proposed incorporation of the upper-layer with powerful super-repository does not aim to overcome other techniques that aim to optimise the degree diameter properties of the structured decentralised environments with larger number of distributed repositories.

Let us assume that there is one repository in the decentralised system in connection with a layer comprising fifteen repositories. Four out of the fifteen repositories with more resources, namely *R3, R20, R37* and *R47*, are chosen to form the upper-layer routing ring. Figure 4.15 illustrates the double-layer routing mechanism, where all the service repositories are sequenced with ID values in clockwise direction. The red circle represents the upper-layer ring with super-repositories and the black circle represents the lower-layer traditional ring [104].

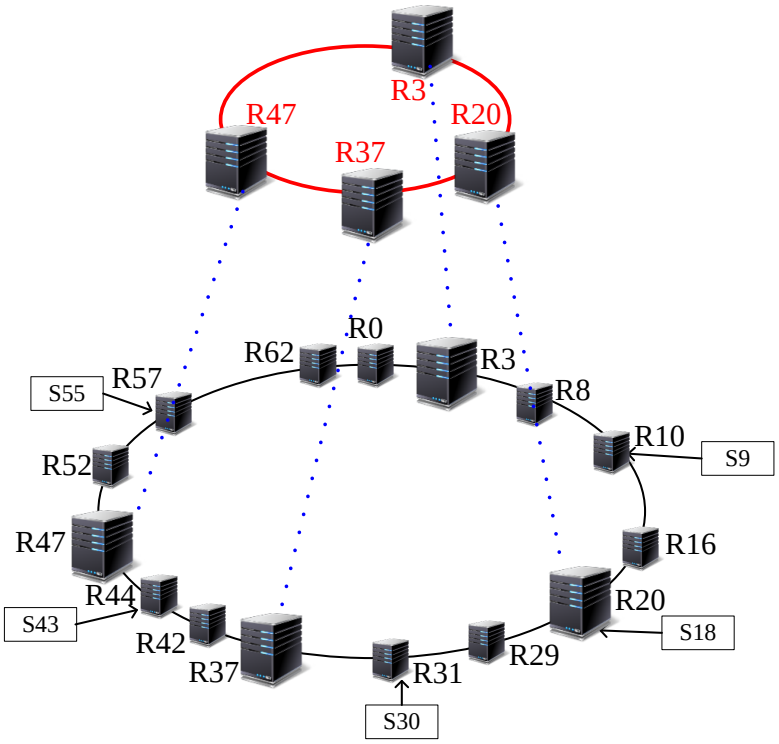

Figure 4.15 Double-layer Routing Mechanism

Similarly, each super-repository maintains a pointer to its lower-layer successor and predecessor repositories. Furthermore, each super-repository should maintain an additional super-routing index in order to route requests on the upper-layer. The definition of clockwise super-routing index is shown in Table 4.5. A super-repository R contains n entries in the super-routing index that divide the interval $[R, R + p^{n-1})$ into n subintervals. The value p is the forwarding power of the super-repository which should be no less than 3. With a forwarding power of p, super-repository can forward a request at a distance of p^i $(1 \leq i \leq log_p 2^m - 1)$ per hop, which is much longer than $2^i (i \leq m)$ which is the case of the lower-layer ring. In order to maintain connectivity with the lower-layer normal repository, the super-repository comprises both the super-routing index and the normal routing index. Normal routing index are promoted in the super-repository to provide the contact points to defer requests

to the lower-layer system. If there are no super-repository identified in the interval, the closet repository $R*$ whose ID is higher than $R + p^{k-1}$ in the lower-layer system will be stored.

Table 4.5 Clockwise Super-routing Index

Notation	Definition
index[k]	First super-repository on circle succeed $(R + p^{k-1}) mod 2^m, 1 \leq k \leq \log_p 2^m,\ 3 \leq p$
index [k].ID	Identifier of the $k-th$ super-repository
index [k].Address	Address of the $k-th$ super-repository or normal repository
super successor	The next super-repository on the identifier circle

Figure 4.16 shows the clockwise super-routing index of an upper-layer super-repository *R3* with a forwarding power of 3. Super-routing index entries $[3, 3 + 3^0)$ and $[3 + 3^0, 3 + 3^1)$ hold the normal repository *R8* because there is no super-repository present within the interval of the index entries. Other entries holding the closest super-repositories are *R20* and *R37* respectively. Similarly, the counter-clockwise super-routing index could be easily generated by cloning clockwise super-routing index. In summary, super-repository *R3* can reach more long distance neighbours such as *R20* and *R37* in one hop enabled by the proposed double-layer rings.

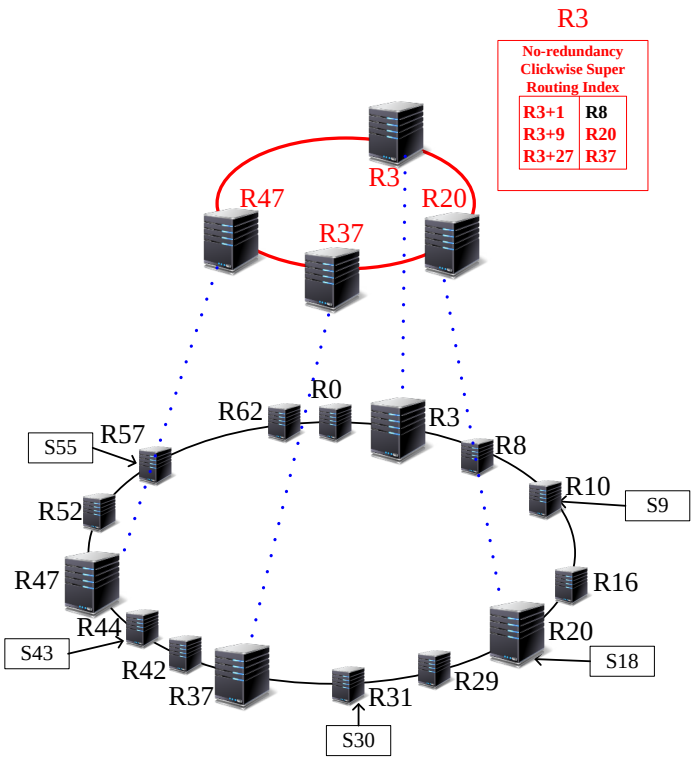

Figure 4.16 Super-repository R3 with Clockwise Super-routing Index

4.4 Double-layer No-redundancy Enhanced Bi-direction Chord

To achieve better performance, the proposed optimised searching algorithm named Double-layer No-redundancy Enhanced Bi-direction Chord (DNEB-Chord) integrates the optimal routing index and the double-layer routing mechanism. Figure 4.17 presents the final routing index of the super-repository *R3* in DNEB-Chord.

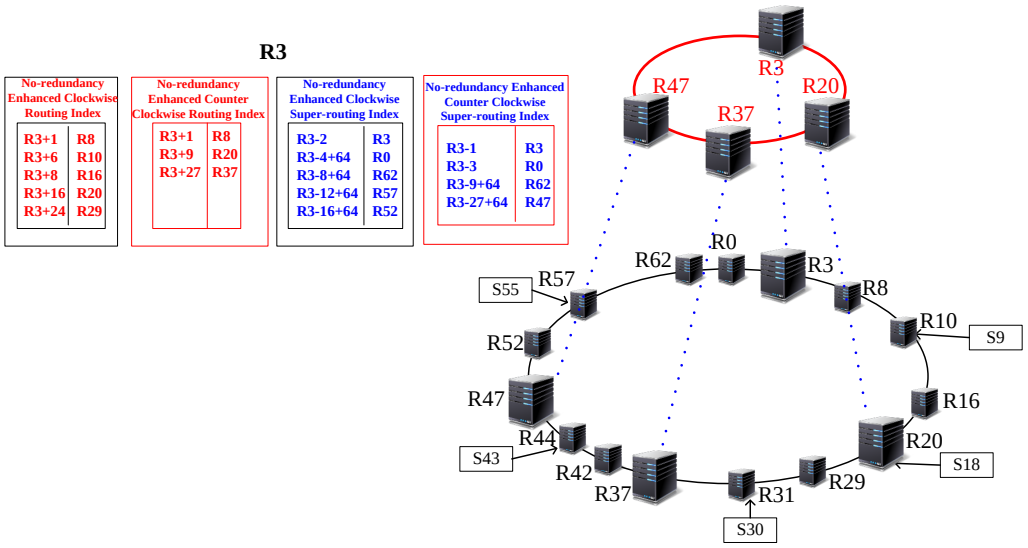

Figure 4.17 Super-repository R3 with DNEB-Chord

Super-repositories routing operation over the upper-layer of DNEB-Chord is similar to normal repositories routing in the lower-layer. As the lower-layer is based on Chord circle, it can guarantee the correctness of the newly proposed upper-layer and the super-repositories routing performance. To super-repository, if the requested service s is not stored in itself, the request is routed in the upper-layer to the neighbour repository R^* whose ID immediately precedes s in the super-routing index. If R^* is a super-repository, the request is continued to be processed in the upper-layer. If R^* is a normal repository in the lower-layer ring, then the request is forwarded to the lower-layer system via R^* to the destination. Hence, normal repositories route requests only in the area where no super-repository exists before the destination.

Since a normal repository in the lower-layer only maintains the lower-layer routing information in their routing index, the resolving process for a lookup request is different from that of the super-repository. To normal repository, the query may be sent to the closest super-repository R^{**} only when the service s

is out of the range of repository *R's* optimal routing index and at least one super-repository *R*** can be reached in one hop. Otherwise, the query will be transferred to the closest normal repository *R** to identify the service s based on its normal optimal routing index.

Therefore, the search operation process with DNEB-Chord can be explained as follows when repository *R* launches the query about an object service s.

a) Step 1: Searching in *R* itself firstly, if the searching is successful, return s. Otherwise, go to step 2.

b) Step 2: Check whether current repository *R* is a super or normal one? If it is a super-repository go to step 3. Otherwise, go to step 5.

c) Step 3: Searching super-repository *R's* optimal super/normal routing index. If s is stored in a one-hop neighbour repository *R**, then return s. Otherwise, go to step 4.

d) Step 4: The query is transferred to the closest repository *R** in *R's* super/normal routing index, the current repository *R* is replaced by *R** and go back to step 2.

e) Step 5: Searching normal repository *R's* optimal routing index. If s is stored in node *R**, then return s. Otherwise, go to step 6.

f) Step 6: If the service s is out of range of *R's* optimal routing index and at least one super-repository *R*** can be found in one-hop, go to step 7. Otherwise, go to step 8.

g) Step 7: The query is transferred to the closest super-repository *R***, *R*** replace the current repository *R* and go back to step 2.

h) Step 8: The query is transferred to the closest normal repository *R** in *R's* optimal routing index, the current repository *R* is replaced by *R** and go back to step 2.

Algorithm 4.6 presents the algorithm for service discovery in decentralised systems based on the proposed DNEB-Chord and DM-index.

Algorithm 4.6: *Service Discovery with DNEB-Chord and DM-index*

```
1   // locating the destination repository with requested service hash ID_s
2   R.find(ID_s)
3   if ID_s located in R
4       return R
5   else if repository R is a super-repository
6           {
7             if ID_s belongs to a one-hop repository R*
8                 return R*
9             else
10              {
11                R*=the closest repository in R's super/normal routing index
12                return R*.find(ID_s)
13                }
14            }
15        else
16            {
17              if ID_s belongs to a one-hop normal repository R*
18                  return R*
19              else if ID_s out the range of R's routing index&& found at least one one-hop super-repository
20                  {
```

21 *R*=the closest super-repository in R's routing index*

22 *return R*.find(ID_s)*

23 *}*

24 *else*

25 *{*

26 *R*=the closest normal repository in R's routing index*

27 *return R*.find(ID_s)*

28 *}*

29 *}*

30 *//discovering service s in the destination repository R with DM-index*

31 $I'_r = Q_p \cap I_r$

32 $P'_{rs} = f_4^{-p}(I'_r)$

33 $C'_{is} = f_3^{-p}(P'_{rs}) \wedge P_{cisi} \subseteq Q_p$

34 $C'_s = f_2^{-p}(C'_{is}) \wedge Q_r \subseteq P_{cso}$

35 $s = D\left(Q_p, Q_r, L, f_1^{-p}(C'_s)\right)$

Figure 4.18 shows an example to search service (*S43)* from repository *R3* with the proposed efficient indexing and searching model (DNEB-Chord and DM-index) in a decentralised environment. In this example, it is evident that repository *R44* is responsible for *S43*. The black arrow shows the routing path in the lower-layer ring, while the red arrow shows the upper-layer routing path in the upper-layer. With the upper-layer routing, repository *R3* routes the request to super-repository *R47* in its super-routing index. Now, the closest repository whose ID precedes 43 in the super-repository *R47's* routing index is the normal repository *R44* and the request is deferred to the underlying system, repository *R44* which is the service (*S43)* successor is finally allocated. Then the satisfied service *S43* for user's request $Q(Q_p, Q_r)$ is dicvoered in repository *R44*.

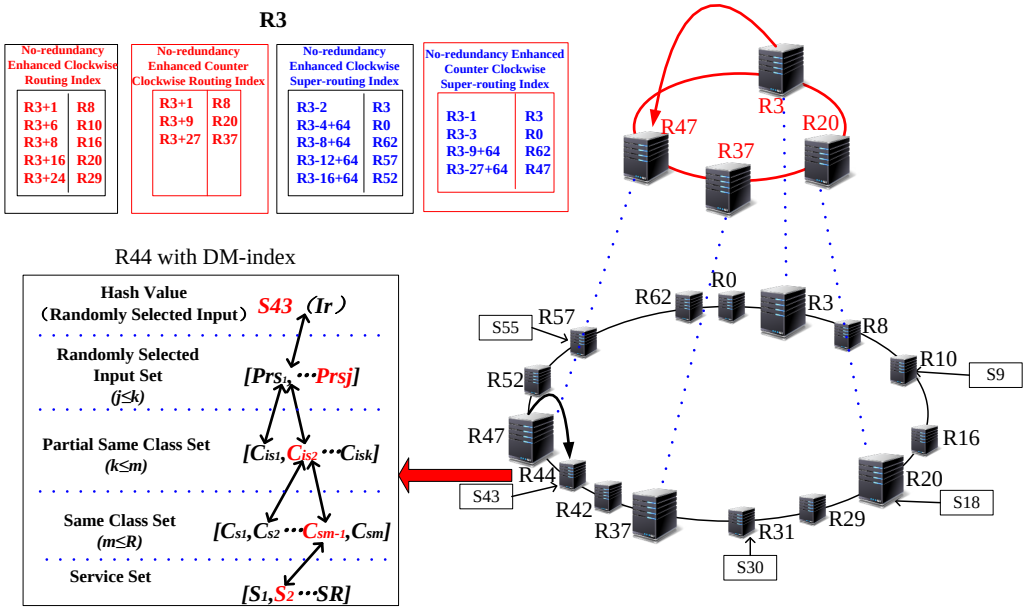

Figure 4.18 Service Discovery with DM-index and DNEB-Chord

In comparison with Figure 4.4, service *S43* can be discovered in fewer hops based on the proposed approach with double-layer routing mechanism and optimal routing index, especially for decentralised systems comprising larger number of repositories with massive stored services. Therefore, the critical contributions of DNEB-Chord are attributed to the process of establishing the efficient double-layer routing mechanism and optimal routing index with less maintenance burden and lower routing cost.

4.5 Summary

As the second main contribution of this research study, this chapter presented the DNEB-Chord searching algorithm including an efficient double-layer routing mechanism and optimal routing index with less maintenance burden and lower routing cost. A set of new index entries are inserted into the middle of the two neighbouring repositories without redundancy, normal repository can

reach more neighbour repositories in one hop. In addition, the upper-layer routing mechanism enables the super-repositories to reach several long-distance neighbour super-repositories in one hop than the lower-layer. Therefore, the hop counts of DNEB-Chord can be significantly reduced. The next chapter evaluates the performance of the proposed novel efficient indexing (DM-index) and searching (DNEB-Chord) model in the newly developed simulation environment.

5 Performance Evaluation

This chapter is aimed at evaluating and analysing efficiencies of the proposed indexing and searching model for service discovery in decentralised environments based on the newly developed DMBSim simulation environment. Three different series experiments are designed to validate the efficiency of proposed DM-index model and DNEB-Chord. Each experiment is detailed along with a discussion of simulation results analysis.

5.1 DMBSim Simulation Environment

In order to exert complete control over the experiments conducted and the statistics captured during the experiment, this thesis developed a novel simulator based on Visual C#, named DMBSim. To evaluate the proposed indexing model, three indexing models are implanted into DMBSim simulator including the sequential index, inverted index and DM-index. Meanwhile, Chord protocol is implanted in DMBSim as the reference standard for DNEB-Chord evaluation. Finally, the efficiency of proposed indexing and searching model including DM-index and DNEB-Chord is evaluated. The statistics of the simulation are written on to a CSV file and then imported into Microsoft Excel for data analysis. The main components of the proposed DMBSim simulator are listed as follows.

a) Graph user's interface

b) Repositories setting up functions

c) Services setting up functions

d) Retrieval requests setting up functions

e) Services retrieval functions

f) Simulation results saving function

An illustration of the control flow in the proposed DMBSim simulator is presented in Figure 5.1.

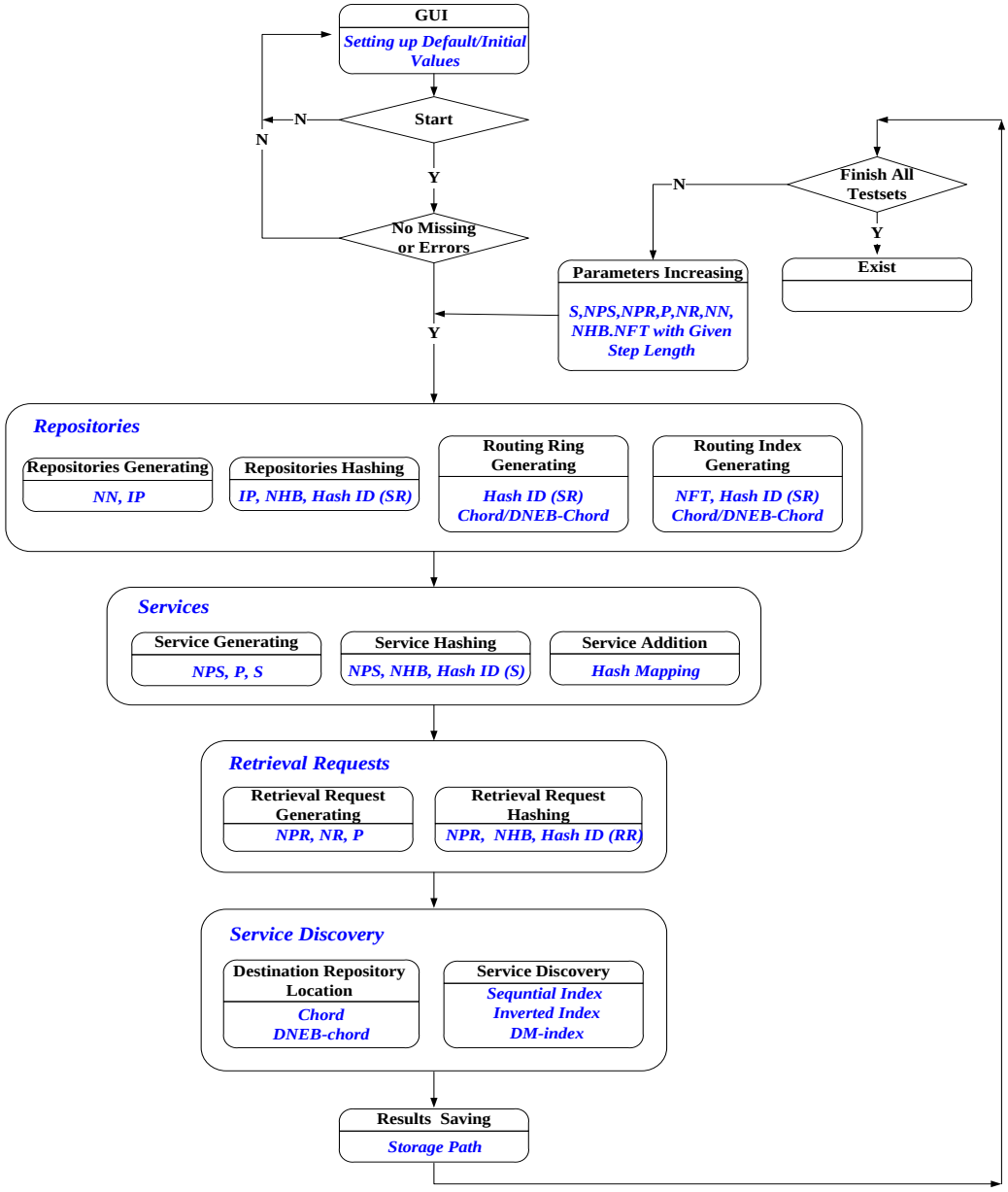

Figure 5.1 DMBSim Flow Diagram

5.2 Experiments Design

Due to the large-scale and decentralised nature of the real-life decentralised environments, it is prohibitively expensive to test an indexing model or a searching algorithm by deploying them on real-world networks performance evaluations [157]. Instead, simulations can be performed mimicking a near-realistic environment to examine the performances of the algorithms without the involvement of costly expenses of creating a real network. In an attempt of achieving a near-realistic simulation environment, this research study developed a new simulator, named DMBSim, by implanting Chod/DNEB-Chord searching algorithms and the sequential index, inverted index and DM-index indexing models for the purpose of validating the proposed indexing and searching model.

The efficiencies of the proposed search algorithm and the indexing model are evaluated under different default and initial values of the simulation parameters including S (number of stored services in the whole decentralised system), P (size of the input parameters pool), NPS/P_i (number of input parameters of each stored service s in repository), NPR/P_r (number of input parameters of each required service), NN (number of repositories in the decentralised system), NR (number of retrieval requests), NHB (length of repository and service hash IDs) and NFT (number of entries in each routing index). Such a varied simulation setting ensures that the performances of the proposed indexing and searching model can be tested under dynamic scenarios by the DMBSim simulator.

The Graphical User Interface (GUI) of the developed DMBSim is shown in Figure 5.2. This GUI allows users to easily set up the default or initial values for all the simulation parameters and iteration times, provides supplements to test the algorithms and protocols under different indexing models and further

provides easily accessible storage directories for saving the simulation results. In this proposed simulator, only one test parameter can be changed and started with new initial value at a time, while all the other network settings remain constant. The code for common settings, layout and configurations could be found in *Appendix 3.1*, *3.2* and *3.3.*

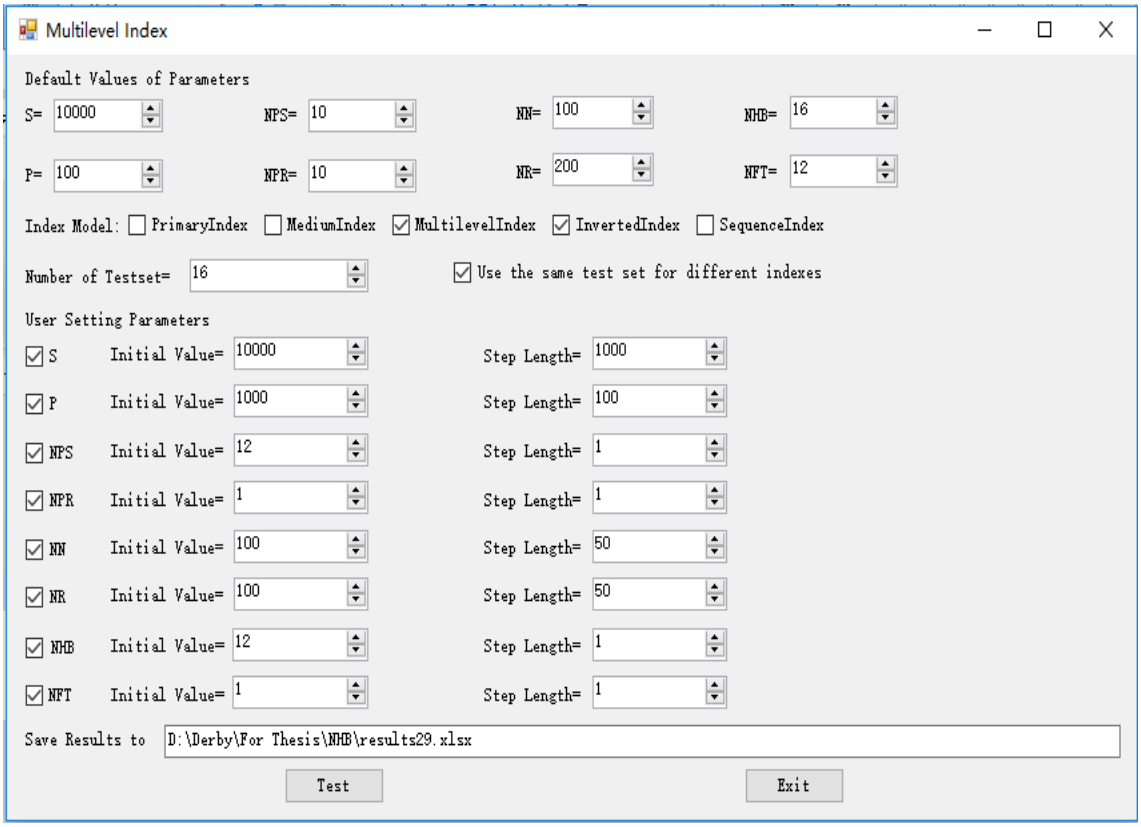

Figure 5.2 GUI of DMBSim Simulator

The experiments conducted in this research are designed in a way that the simulation presents a significant number of properties to analyse, combinations of parameters to experiment with and scenarios to deploy in the simulation. The code for simulation results could be found in *Appendix 3.10 and 3.11*. The most pertinent experiments conducted in this research are summarised as follows. In the first series of experiments, the count of the retrieved services will be tested and analysed to validate the efficiency of the proposed DM-index model. In the

second experimental scenario, the hop count will be considered to evaluate the efficiency of the proposed DNEB-Chord algorithm. Finally, the retrieval time will be focused to evaluate the efficiency of the proposed novel efficient indexing and searching model for service discovery in decentralised environments.

5.3 DM-index Model Validation

Experiments in this section are conducted to evaluate the performances of the DM-index model by validating the correctness of the theoretical analysis of expectation of the traversed services count, discussed earlier in Chapter 3. Therefore, the primary objective of this experiment is to test whether the experimental results are supporting the prior algebraic analysis of expectation or not. The code for sequential index, inverted indexand DM-index model could be found in *Appendix 3.4, 3.5, 3.6, 3.7 and 3.8*. The expectation of the traversed services for the sequential index (E_s), inverted index (E_i), primary deployment model (E_{pr}), partial deployment model (E_{pt}) and full deployment DM-index (E_{fl}) have been analysed in Chapter 3. To this end, the efficiencies of the proposed full deployment model of the DM-index are evaluated against the sequential and the inverted index.

From the prior formulas discussed in Chapter 3, the expectation of the traversed service count of the sequential index, inverted index and DM-index are dependent on S, P, $NPS(P_i)$ and $NPR(P_r)$. Therefore, four different experiments are conducted to verify the prior algebraic analysis.

Since the expectation of the sequential index model is simple and obvious, the sequential index is chosen as the reference standard for all the experiments. The dataset used for the initial validation experiments are synthetic according to

different parameters. The default and initial values of the simulation parameters, step lengths and the number of iteration are shown in Table 5.1. In order to extract a significant number of simulation results for guaranteeing precision, a single experiment is set to run for 20 times and the results are averaged accordingly.

Table 5.1 Configurations for DM-index Validation

Parameter	Default/Initial Value	Step Length	Number of Iterations
S	10000	1000	16
NR	200	/	/
P	100	10	16
$NPS(P_i)$	10	1	16
$NPR(P_r)$	10	1	16
NFT	12	/	/
NHB	16	/	/
NN	100	50	/

5.3.1 Impact of the Number of Stored Services

In order to evaluate the impact of the number of stored services, the value of S is changed from 10000 through to 26000 with an increase of 1000 in each iteration and all other parameters remain constant as listed in Table 5.1. Figure 5.3 presents the results of this experiment for the three indexing models respectively.

The expectation of the sequential index $E_s = S \times NR$ is simple and directly proportional to S. From Figure 5.3, the obtained experiment results are very similar to the expectation of $E_i = \frac{P_i \times P_r}{P} \times S \times NR$ and are increasing in proportional to S as expected. The expectation $E_{fl} = |Q_p \cap I_r| \times \frac{\overline{|C_s|}}{|C_{ts}|} \times NR$ looks unreverent with S. Since the number of $\frac{\overline{|C_s|}}{|C_{ts}|}$ is impacted by S, E_{fl}

increases accordingly depending on S. These results are in accordance with the theoretical analysis and further demonstrate that the DM-index model traverses much fewer number of services than those of the sequential index and inverted index models, despite the increase witnessed in the value of S.

The critical issue of this experiment is that the validation results confirm prior algebra analyse result: DM-index traverses much fewer services than the sequential index and inverted index when S keeps changing.

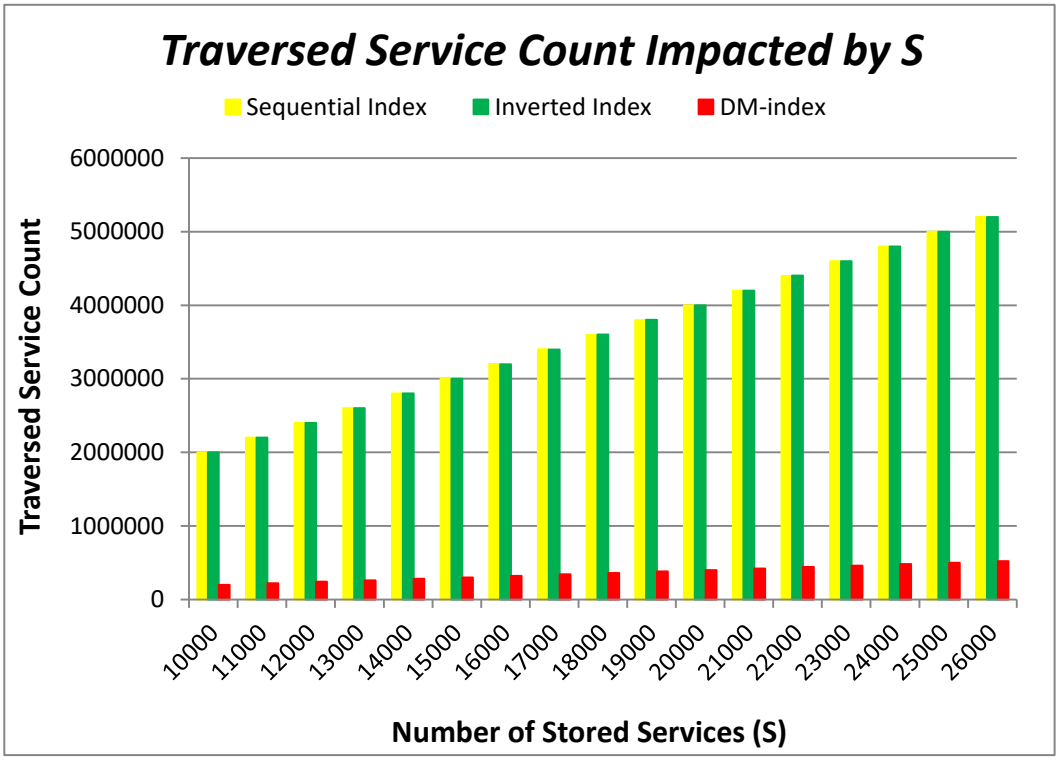

Figure 5.3 Traversed Service Impacted by S

5.3.2 Impact of the Number of Input Parameters of Each Stored Service

To evaluate the impact of the number of input parameters of each stored service, the value of NPS/P_i is changed from 10 to 26 with an increase of 1 in every

iteration and all the other parameters remain constant. Figure 5.4 presents the results of this experiment for the three models respectively.

Since E_s is dependent only on the number of service S, experimental validation proves that E_s remain constant despite the changes witnessed in the value of NPS/P_i. Experimental results further confirm that E_i is directly proportional to NPS/P_i. Since NPS/P_i increases with constant S and there is a possibility further reduction in the value of $\frac{\overline{|C_s|}}{|C_{ts}|}$, the obtained results prove that NPS/P_i does not affect E_{fl} to any notable level.

Similar to the first experiment, the DM-index model traverses much fewer number of services than those of the sequential index and inverted index models when an increase is witnessed in the value of NPS.

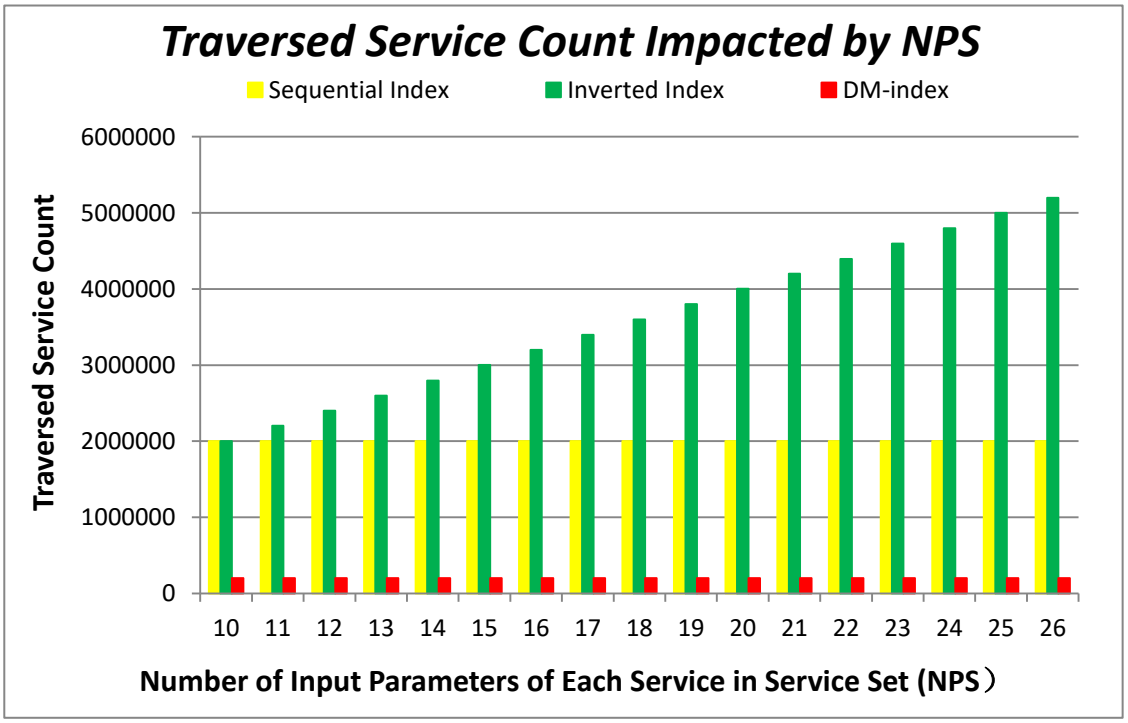

Figure 5.4 Traversed Service Impacted by NPS

5.3.3 Impact of the Number of Input Parameters of Each Retrieved Service

To evaluate the impact of the number of input parameters of retrieved services in the third experiment, the value of NPR/P_r is changed from 10 to 26 with an increase of 1 in every iteration and all the other parameters remain constant. Figure 5.5 presents the results for the three models respectively.

The experiment results confirm that E_s is behaving the same way as per the expectation of the theoretical verification. The way that both E_i and E_{fl} impacted by NPR/P_r is very similar to the theoretical expectations. Similar to the two previous experiments, it can again be concluded that the proposed DM-index model achieves better retrieval efficiency than that of the sequential index and inverted index models, despite the changes witnessed in the value of NPR.

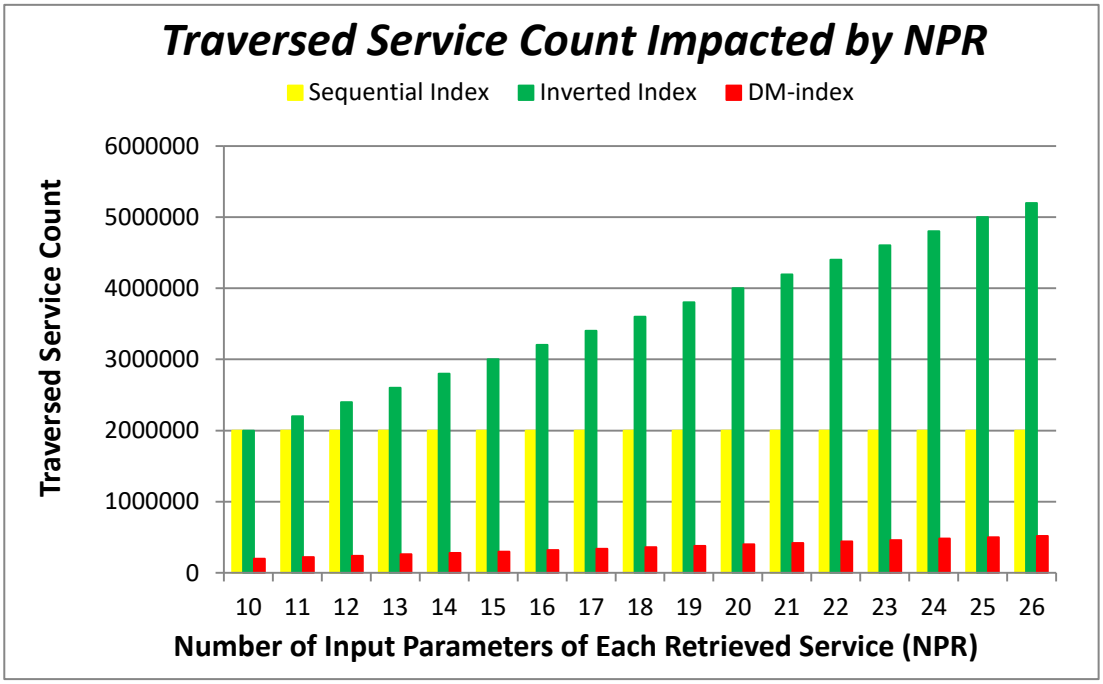

Figure 5.5 Traversed Service Impacted by NPR

5.3.4 Impact of the Size of the Input Parameters Pool

In the final simulation of the first series of experiments, the impact of the input parameter pool P is analysed by changing the value of P from 100 through to 260 with an increase of 10 in each iteration. Figure 5.6 presents the results of the experiments for the three models respectively. Experiment results confirm that the changes in the value of P does not affect E_s, further E_i and E_{fl} are inversely proportional to P. Thus, it can be concluded that the proposed DM-index model is outperforming both the sequential and inverted index models.

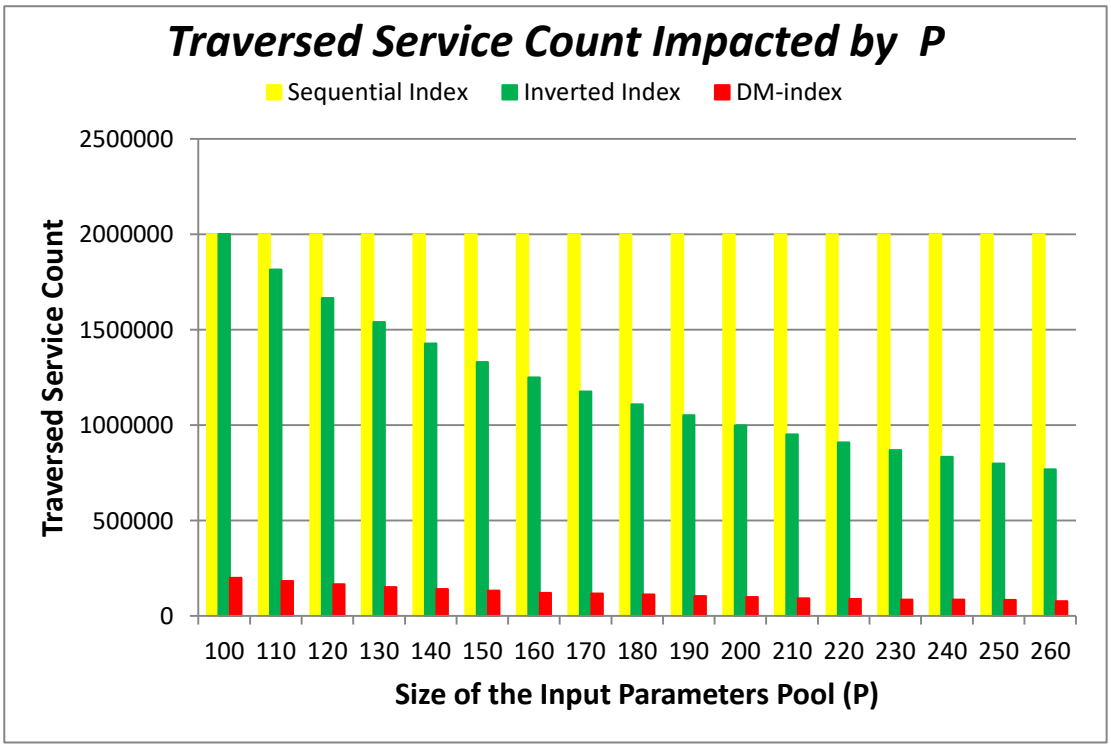

Figure 5.6 Traversed Service Impacted by P

5.3.5 DM-index Validation Results Analysis

From the obtained experimental results, it is evident that the proposed DM-index model can significantly reduce the number of traversed services and

can achieve better performance than the inverted index. The first series of experiments demonstrate that the DM-index model can effectively reduce the number of traversed services whilst retrieving user-desired services, reducing around 80% of the services traversed by the sequential index and the inverted index models in most circumstances. However, the DM-index model may not be completely suitable for small-scale service repositories, since smaller repositories usually comprise only a fewer number of services characterising similar input and output parameters.

5.4 DNEB-Chord Algorithm Validation

This section aims to evaluate the performance of the optimised DNEB-Chord algorithm against the traditional Chord under the same host configurations. The performances of both the searching algorithms are evaluated in terms of the achieved hop count. Usually, a lower hop count reflects that only a fewer number of repositories are required to be traversed before finding the required services. The code for DNEB-Chord could be found in *Appendix 3.9*.

Before starting the validation of the proposed DNEB-Chord algorithm, there are a few issues to be described. Firstly, whilst locating the service for a given query, the traversed number of repositories or the hop counts in a decentralised system is only influenced by the distance between the start repository and the destination repository. This means that the hop count only reflects the number of NHT, $NN\ NR$ and NFT. Secondly, the indexing model deployed in the service repository does not affect the hop count, so only the DM-index model will be considered in the evaluation of the DNEB-Chord algorithm in this section.

The initial values of the simulation parameters are marginally different from the previous experiments, as shown in Table 5.2. Similar to the DM-index validation, every simulation experiment is set to run for 20 times and the obtained results are averaged to guarantee accuracy.

Table 5.2 Configurations for DNEB-Chord Algorithm Validation

Parameter	Default Value	Initial Value	Step Length	Number of Iterations
S	10000	10000	2000	16
NR	200	200	50	16
P	100	100	/	/
$NPS(P_i)$	10	10	/	/
$NPR(P_r)$	10	10	/	/
NFT	12	1	1	9
NHB	16	9	1	9
NN	100	100	20	16

5.4.1 Impact of the Number of Stored Services

Although the hop count should not be affected by S, Figure 5.7 depicts the total number of hop counts when retrieving 200 services, for different values of S changing from 10000 through to 42000. It can be observed that the DNEB-Chord only requires around 400 hop counts to retrieve all the required services. Meanwhile, the Chord protocol requires around 800 hop counts to find the same services. This experiment demonstrates that the DNEB-Chord can optimise the average hop count by around 50% than that of the Chord protocol, due to its better resource allocation management availed by the optimal routing index and routing mechanism..

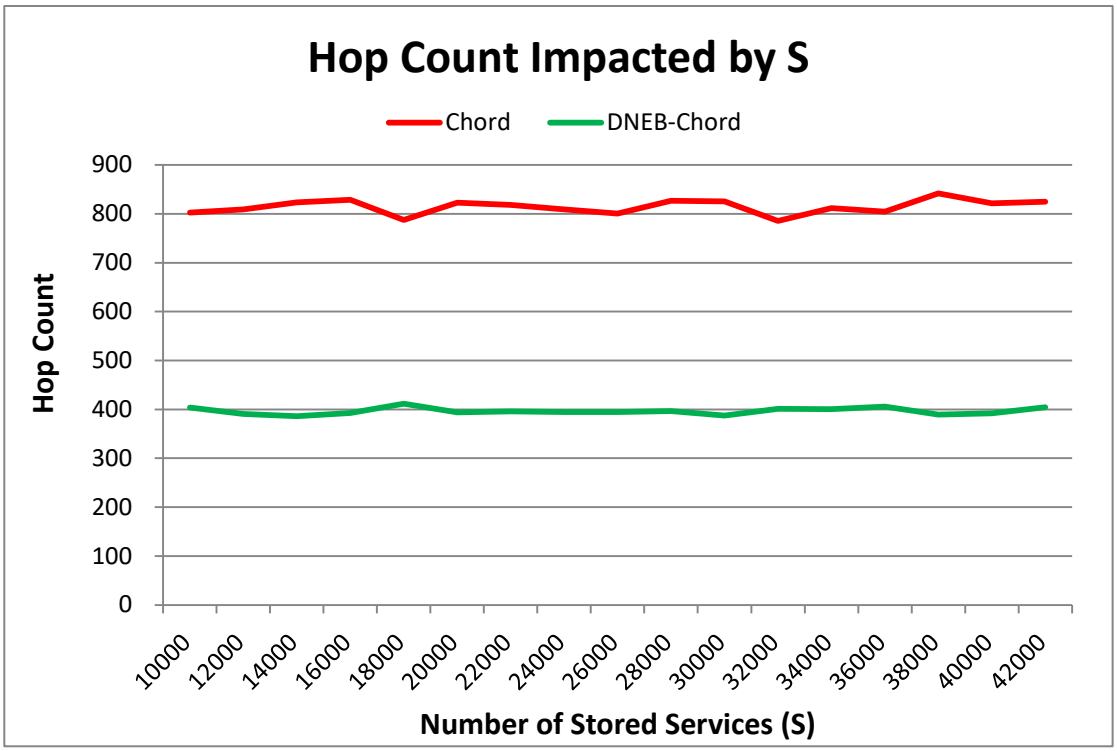

Figure 5.7 Hop Count Impacted by S

5.4.2 Impact of the Number of Retrieval Requests

Secondly, the hop count is measured under different NR values. Figure 5.8 represents the incurred number of hop counts for the two protocols, when the NR value is changed from 100 through to 900 while the other parameters remain constant. It can be observed that the hop counts of both the DNEB-Chord and Chord are exhibiting increasing trends with increasing values of NR. However, the increasing trend of DNEB-Chord is much lower than that of Chord. Therefore, from the simulation results, it is evident that DNEB-Chord exhibits a lower hop count in comparison to Chord under different values of NR.

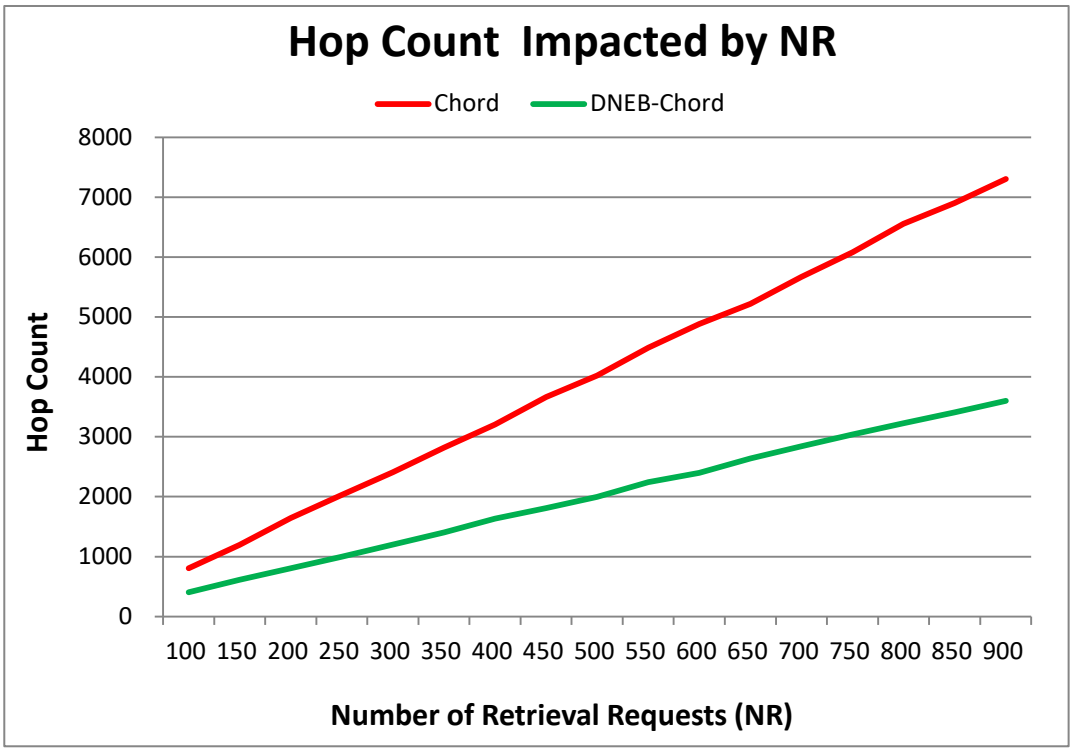

Figure 5.8 Hop Count Impacted by NR

5.4.3 Impact of the Length of Repository and Service Hash ID

In the third experiment, the hop count is analysed under different values of NHT. In Figure 5.9, the hop counts of both DNEB-Chord and Chord are exhibiting an increasing trend for a corresponding increase in the values of NHT. However, DNEB-Chord characterises much lower hop count than that of Chord, for the corresponding values of NHT, which proves that DNEB-Chord can identify the required services with reduced number of hops than Chord, achieving an average of 50% reduction in the incurred hop counts.

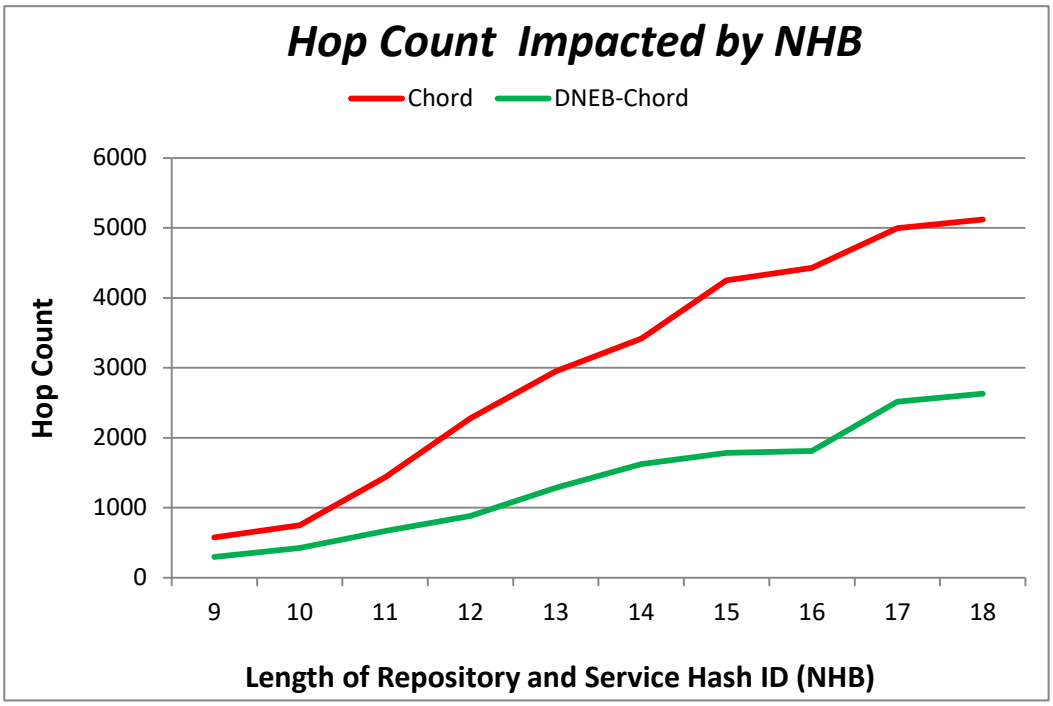

Figure 5.9 Hop Count Impacted by NHB

5.4.4 Impact of the Number of Repositories

When the number of repositories becomes more substantial, more repositories will join the decentralised system, whereby the distance between the repositories may become short. Furthermore, services may also be transferred to the newly joining repositories. This is the reason why Figure 5.10 exhibits a marginally decreasing trend in the number of hop counts, when there is an increase in the values of NN. Still, the number of incurred hop counts are maintained at the lowest possible level by the DNEB-Chord under increasing number of repositories..

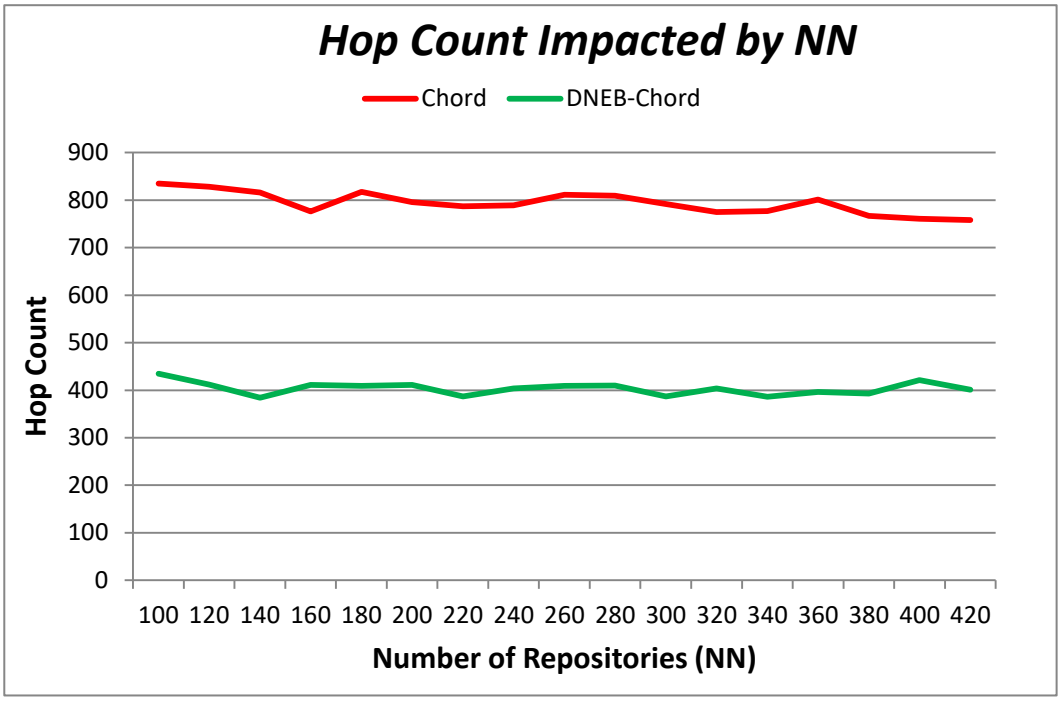

Figure 5.10 Hop Count Impacted by NN

5.4.5 Impact of the Number of Entries in Each Routing Index

Now, the impacts of NFT over the hop counts are evaluated for DNEB-Chord and Chord accordingly. NHB is set to 12 in this experiment and so the range for NFT only can be $1 \leq NFT \leq NHB$. Both Chord and DNEB-Chord exhibits a decreasing trend in the required number of hop counts to reach the requested services with a corresponding increase in the values of NFT, as shown in Figure 5.11. This is due to the fact that each repository maintains more entries which enable them to reach other repositories with fewer hops. At some point, the difference between Chord and DNEB-Chord becomes narrow when NFT increases further. Thus, it is evident that DNEB-Chord can achieve better search performances than Chord especially when nodes maintain fewer entries.

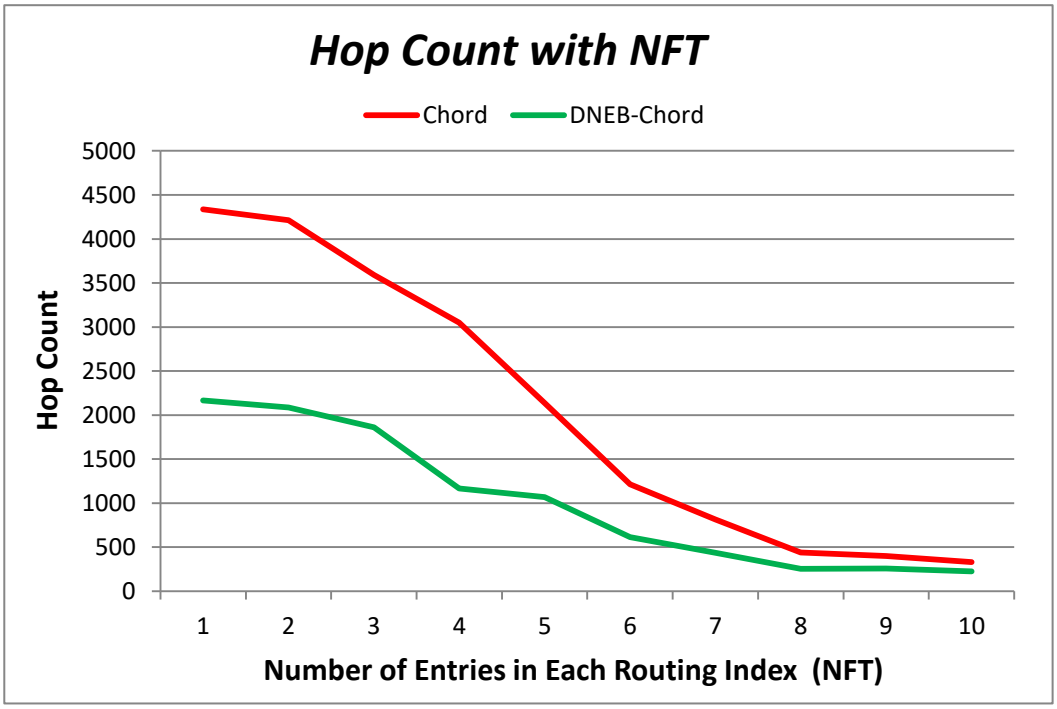

Figure 5.11 Hop Count Impacted by NFT

5.4.6 DNEB-Chord Validation Results Analysis

From all the experiment results, it is evident that the DNEB-Chord outperforms the Chord protocol, by effectively reducing the incurred hop counts by an average of 50% than that of Chord. This demonstrates that DNEB-Chord outperforms Chord under the same network configurations. Thus, the proposed DNEB-Chord searching algorithm can achieve better service discovery performances than the traditional Chord protocol.

5.5 Service Discovery Efficiency Validation

This section presents a couple of simulation scenarios to evaluate the service discovery efficiencies of the proposed searching model integrated with the developed indexing model in decentralised environments. To demonstrate the efficiency of the proposed approach, retrieval time is analysed in this section.

The initial values of the simulation parameters are similar to the previous experiments, as shown in Table 5.3 and every experiment is set to run for 20 times and the obtained results are averaged to guarantee accuracy.

Table 5.3 Configurations for Service Discovery Efficiency Validation

Parameter	Default/Initial Value	Step Length	Number of Iterations
S	10000	2000	16
NR	100	50	16
P	100	50	16
$NPS(P_i)$	10	5	16
$NPR(P_r)$	10	/	/
NFT	12	/	/
NHB	16	/	/
NN	100	/	/

5.5.1 Impact of the Number of Stored Services

Figure 5.3 and Figure 5.7 have previously shown that the number of traversed services is increasing in proportional to S and the hop count should not be affected by S. Figure 5.12 and Figure 5.13 depict the total retrieval time for retrieving 100 services under a changing value of S value from 10000 through to 42000. Figure 5.12 indicates that the proposed DNEB-Chord can achieve better retrieval efficiency than Chord in terms of the incurred retrieval time. It can further be observed from Figure 5.13 that the proposed approach only consumes less than *6ms* to retrieve all the services in any circumstances. Meanwhile, the inverted index and Chord protocol characterise significantly higher retrieval time than that of the proposed DM-index and DNEB-Chord, witnessed at 25-30 times more than that of proposed approach whilst finding the same set of services. From both Figure 5.12 and Figure 5.13 it, can be observed that the retrieval time of the DM-index with DNEB-Chord model, inverted index model with Chord and inverted index with DNEB-Chord show

an increasing trend in the incurred retrieval time for increasing values of S. This demonstrates that the proposed indexing and searching model can optimise the efficiency of service discovery in decentralised systems by around 90%, due to its better service maintenance provided by the DM-index and the better repository location management facilitated by DNEB-Chord.

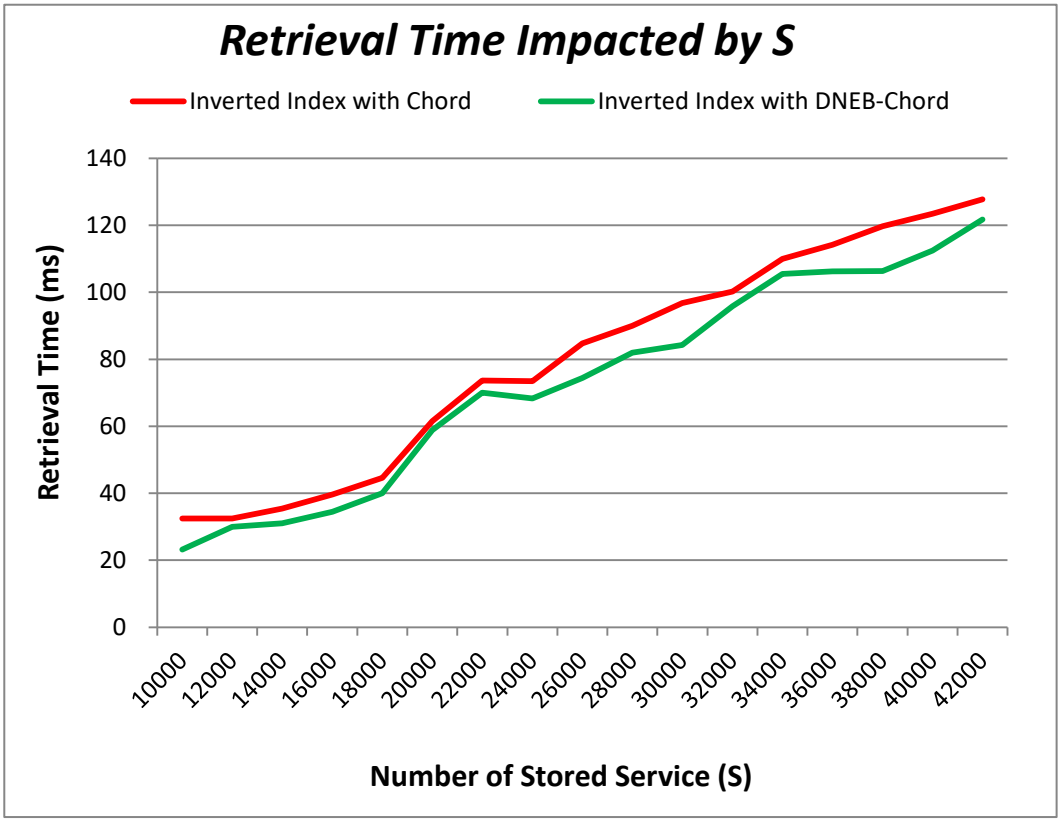

Figure 5.12 Retrieval Time Impacted by S: Case 1

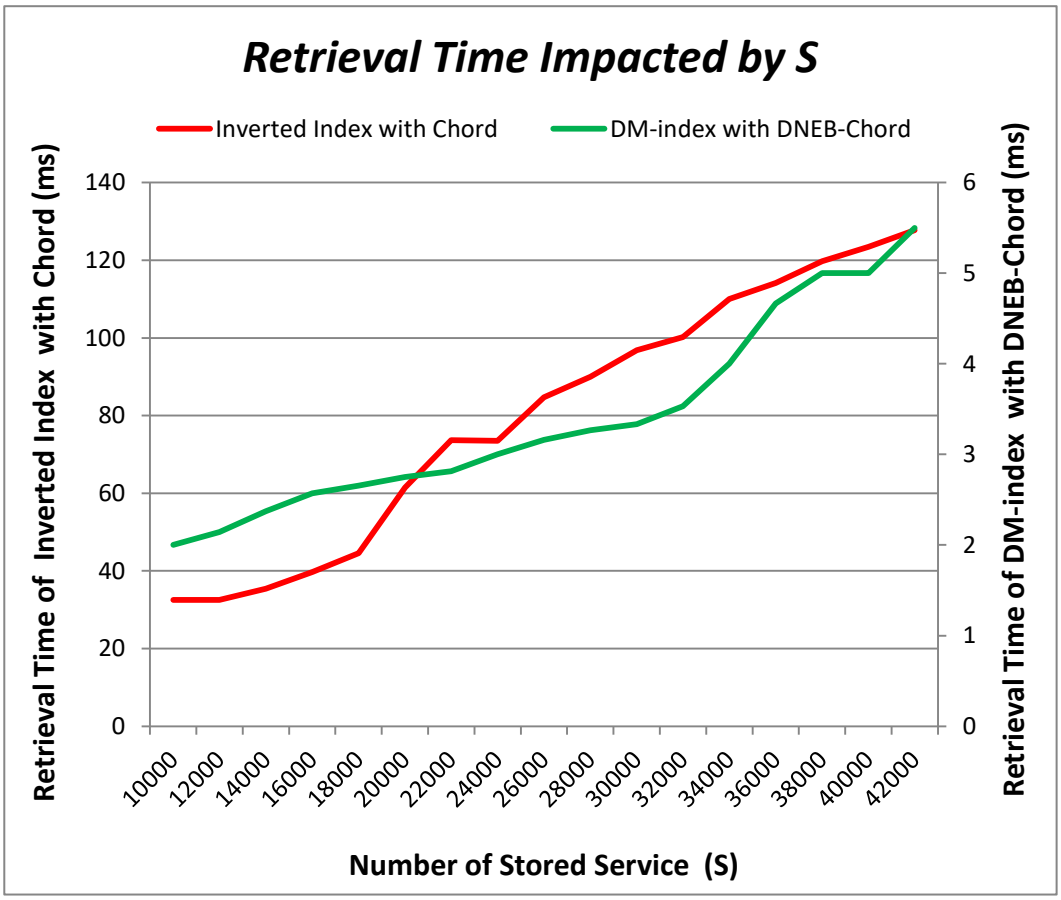

Figure 5.13 Retrieval Time Impacted by S: Case 2

5.5.2 Impact of the Size of Input Parameters Pool

In the second experiment, the impact of the input parameter pool P is analysed by changing the value of P from 100 through to 900 with an increase of 50 in each iteration. Figure 5.14 indicates that the inverted index model with DNEB-Chord model is more efficient than the inverted index model with Chord. Figure 5.15 presents that the changes in the value of P does not affect the efficiency of the proposed approach very much. Further the retrieval time of the inverted index model with Chord is inversely proportional to P. Thus, it can be concluded that the proposed DM-index with DNEB-Chord searching model is outperforming inverted index model with Chord.

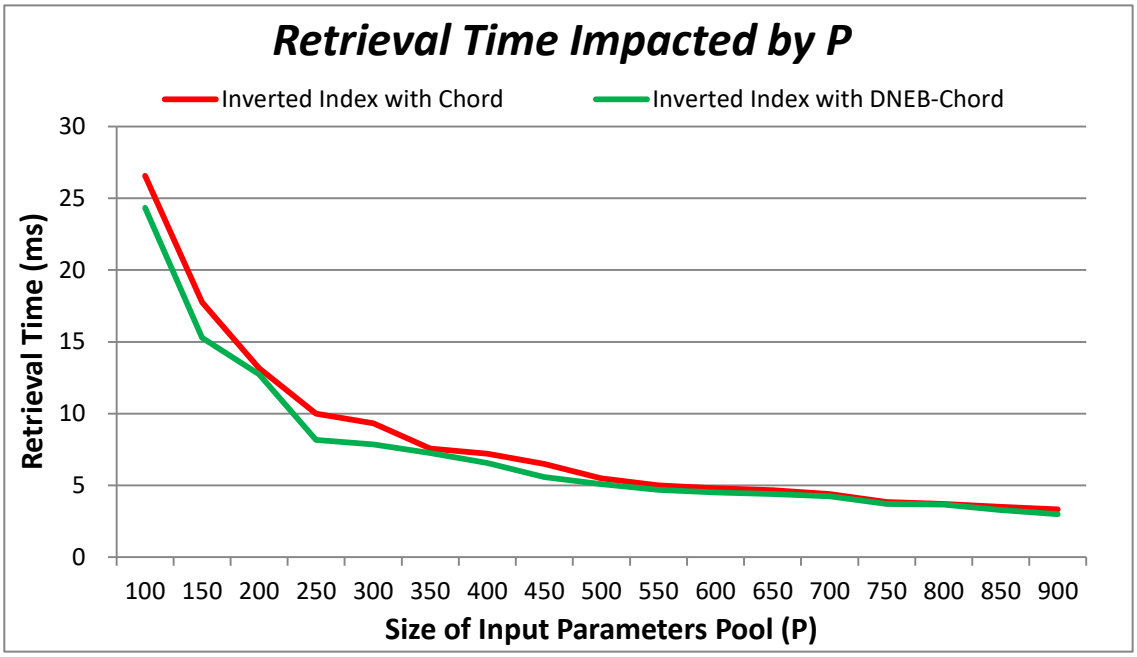

Figure 5.14 Retrieval Time Impacted by P: Case 1

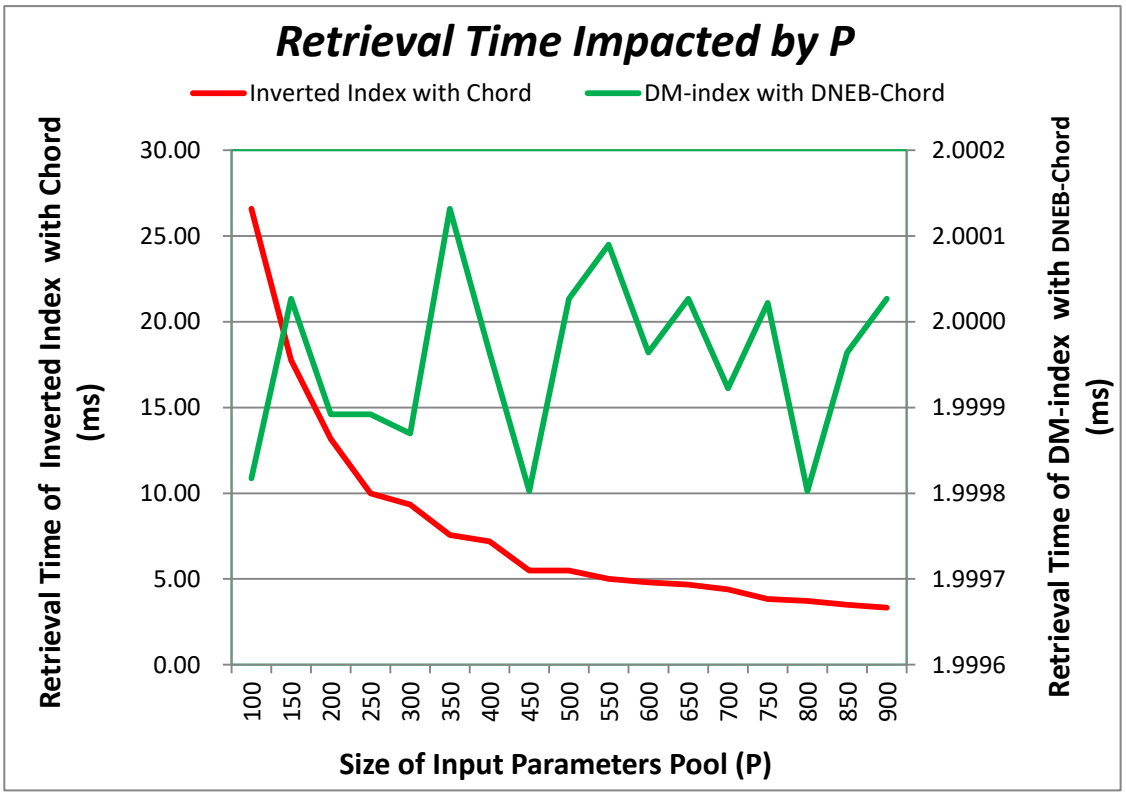

Figure 5.15 Retrieval Time Impacted by P: Case 2

5.5.3 Impact of the Number of Input Parameters of Each Stored Service

To evaluate the impact of the number of input service parameters of each stored service, the value of NPS/P_i is changed from 10 through to 90 with an increase of 5 in every iteration and all the other parameters remain constant. From Figure 5.16, it is evident that DNEB-Chord is exhibiting a higher increase trend than that of Chord in terms of the retrieval time, when the number of input parameters increases beyond 50. This may suggest that Chord is exhiitng a better retrieval time efficiency than DNEB-Chord because of its less maintaience burden. The results of this experiment presented in Figure 5.17 further confirm that the retrieval time of two models are directly proportional to NPS/P_i. Similar to the prior experiments, the proposed efficient indexing and serching model costs much less time than the inverted index model with Chord when there is an increase in the value of NPS/P_i.

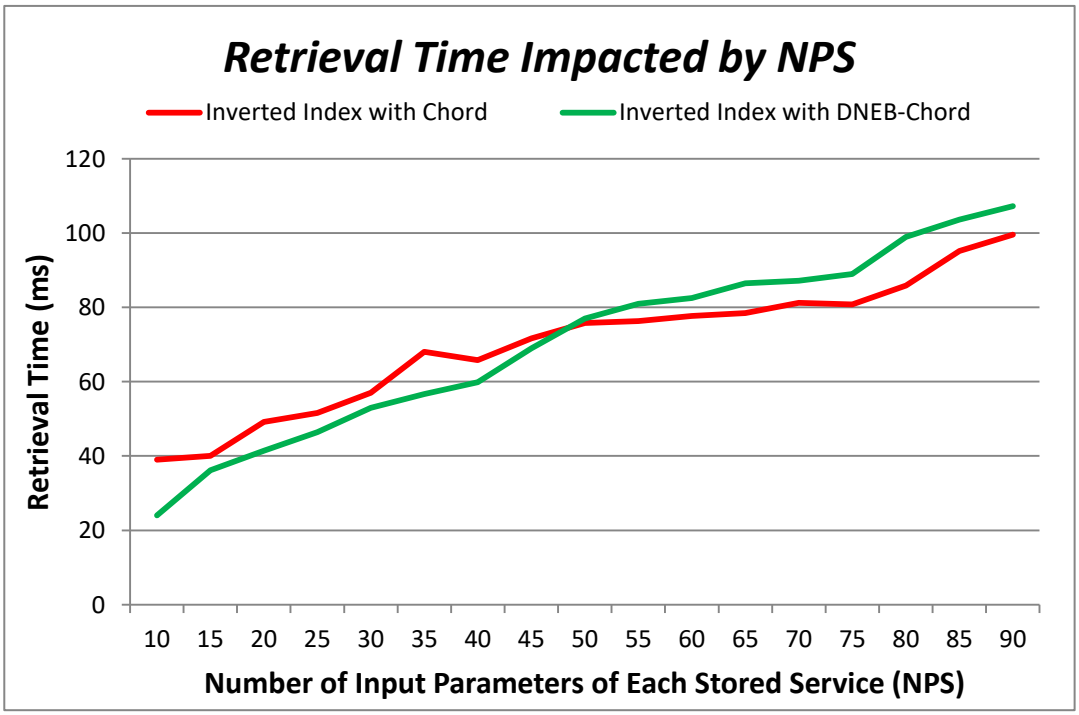

Figure 5.16 Retrieval Time Impacted by NPS: Case 1

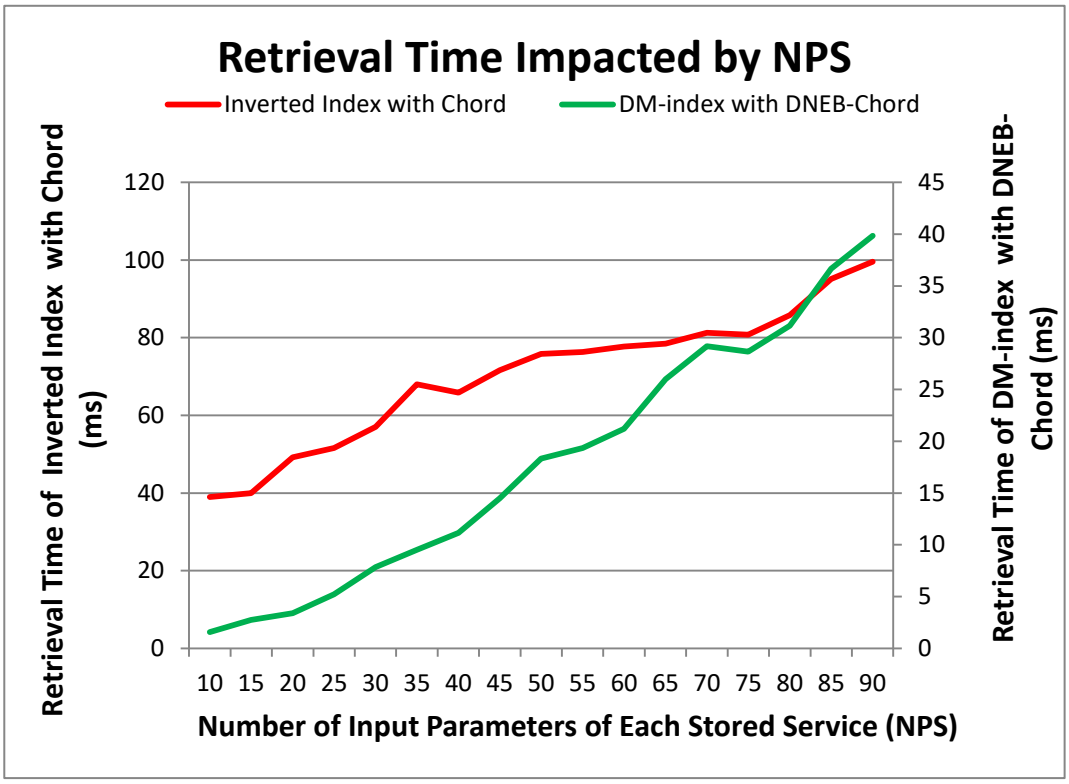

Figure 5.17 Retrieval Time Impacted by NPS: Case 2

5.5.4 Impact of the Number of Retrieval Requests

Finally, the retrieval time is measured under different NR values respectively. Figure 5.18 and Figure 5.19 represents the incurred number of retrieval time when the NR value is changed from 100 through to 900 while the other parameters remain constant. Figure 5.18 shows similar results to Figure 5.12 and Figure 5.14. It can be observed from Figure 5.19 that the retrieval time of both the DM-index model with DNEB-Chord and the inverted index model with Chord are exhibiting increasing trends for increasing values of NR. However, the retrieval time of the proposed approach is much lower than that of the inverted index model with Chord. Therefore, from the simulation results, it is evident that the DM-index model with DNEB-Chord protocol can achieve

better retrieval time efficiency for service discovery in decentralised environments

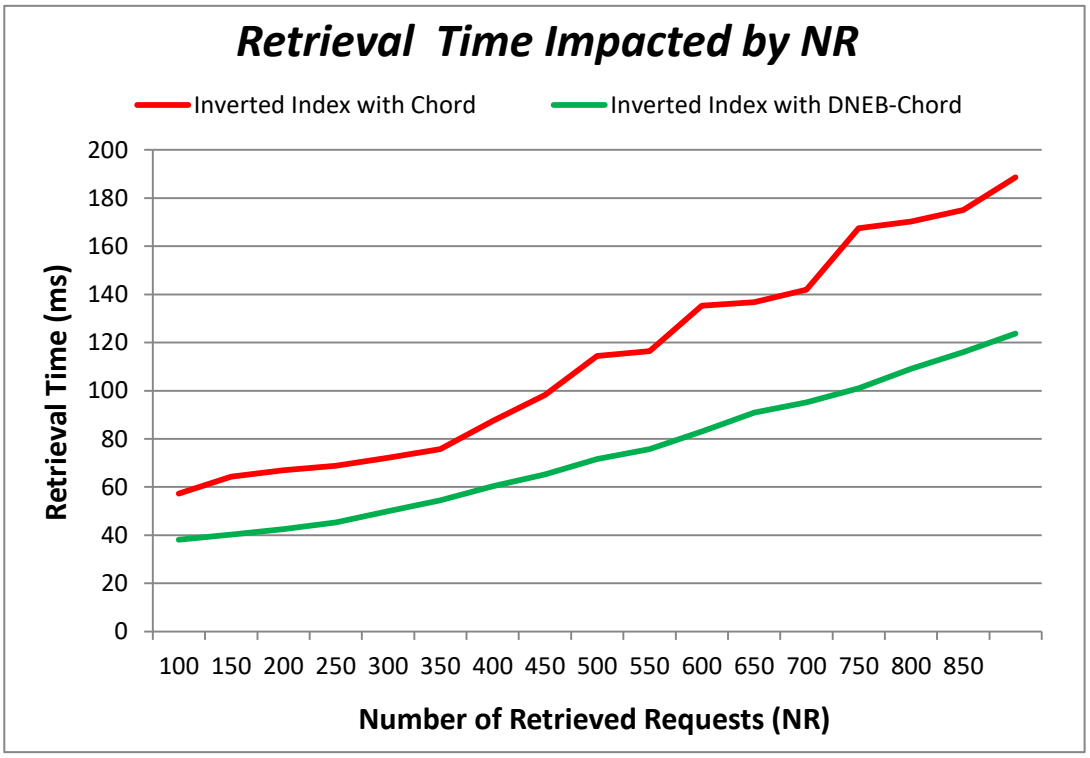

Figure 5.18 Retrieval Time Impacted by NR: Case 1

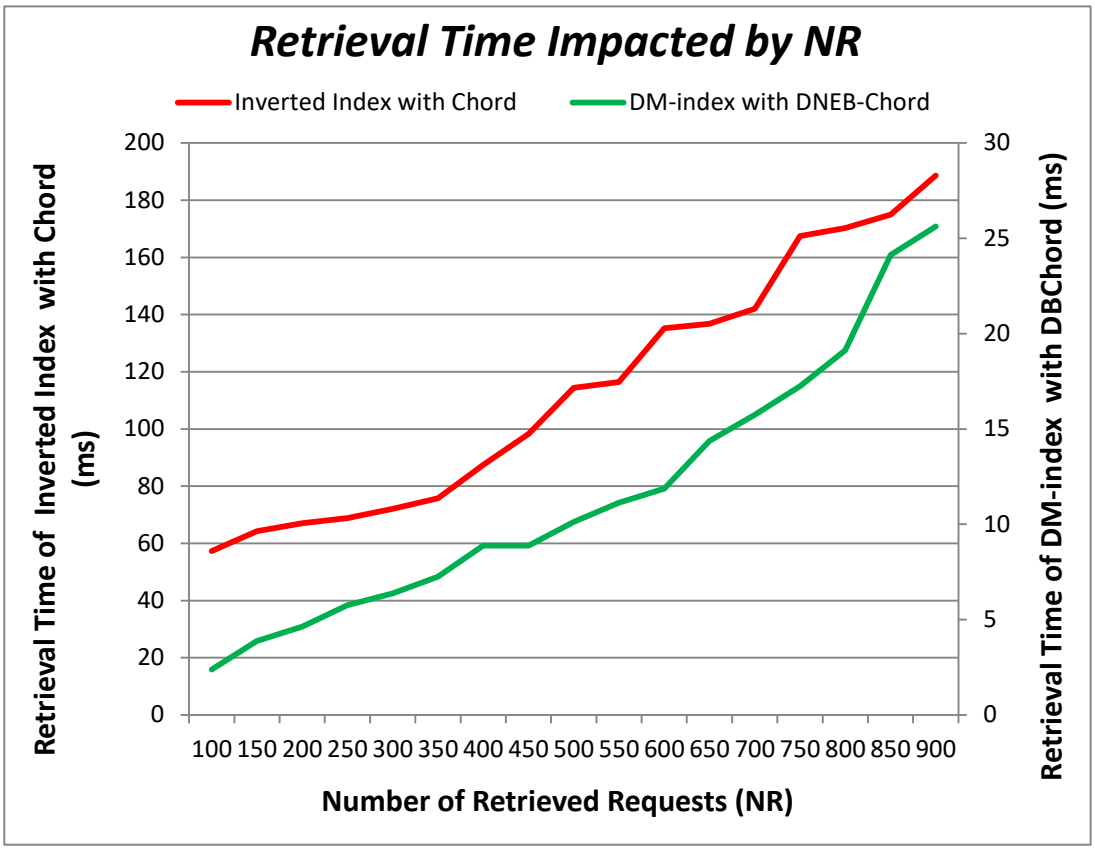

Figure 5.19 Retrieval Time Impacted by NR: Case 2

5.5.5 Efficiency Validation Results Analysis

To conclude, this section conducted experiments considering the service retrieval time in decentralised systems to evaluate the performances of the DM-index model with DNEB-Chord against the inverted index model with Chord. Experimental evaluation results show that the developed DM-index model and the DNEB-Chord algorithm can reduce the retrieval time and achieve efficient service discovery in decentralised environments comprising numerous service repositories with massive stored services.

5.6 Summary

This chapter presented the performance evaluation of the developed indexing and searching model through a range of experiments conducted in various simulation settings. The experiments are conducted in three phases: firstly evaluating the efficiency of the proposed DM-index model in terms of the number of traversed services whilst finding the user desired services; secondly evaluating the performance of the developed DNEB-Chord protocol in terms of the incurred hop count whilst locating services and thirdly evaluating the integrated implementation of the proposed efficient indexing and searching model in terms of the service retrieval time. All the conducted series of experiments demonstrate that the proposed indexing and searching model performs significantly better than the inverted index model and the traditional Chord protocol accordingly, in a decentralised system comprising large number of distributed repositories with massive stored services. The next chapter concludes this thesis along with discussing the limitations and outlining the future research directions.

6 Conclusions and Future Research Directions

This section discusses the contributions and future research directions of this study. The DM-index model and DNEB-Chord algorithm are two major contributions of this research work. The experimental analysis based on DMBSim highlights the contributions and the productive results. Finally, the future research directions present a discussion of the extension of this work.

6.1 Major Contributions of the Thesis

The research objectives of this thesis are achieved through a combination of three core components such as the efficient indexing analytics, efficient allocation analytics and reasonable simulation analytics.

The first contribution of this thesis is identifying the research gap and uncovering the hidden barriers to achieve efficient service discovery in the decentralised environment comprising large number of service repositories with massive stored services, along with exploring the research gaps of existing state-of-the-art research techniques based on extensive literature study conducted in Chapter 2.

The second contribution of this thesis is the DM-index mode which has been presented in Chapter 3. This indexing model is demonstrated in theoretical analysis and experimental validation. The efficiency of service discovery in distributed service repositories with DM-index model outperforms the inverted index and sequential index. It encompasses some scenarios to show the variation in different parameters set. The following considerations have been raised during the experimental analysis and listed to represent the significant conclusions.

a) Firstly, the redundancy information in service repositories could be eliminated and the searching space could be narrowed.

b) Secondly, both theoretical and experimental validation results show that the count of traversed services and retrieval time for the DM-index model much better than the sequential index and inverted index.

c) Finally, it concludes that the proposed DM-index model could achieve efficient service discovery in decentralised environments comprising massive stored services.

The third contribution of this thesis is the optimised DNEB-Chord algorithm along with double-layer routing mechanism and optimal routing index which has been presented in Chapter 4. The performances of proposed DNEB-Chord algorithm are demonstrated in the experimental analysis. It encompasses some different cases to show hop count impacted by different submissions. The following list represents the significant conclusions.

a) Firstly, distributed repositories could reach many more neighbours in one hop with the proposed double-layer routing mechanism and optimal rouging indexes.

b) Hop count to reach the objective repository could be reduced about 50% by deploying DNEB-Chord in the decentralised system consists of large number of distributed service repositories.

c) Finally, it concludes that the proposed DNEB-Chord could achieve efficient service discovery in decentralised environments comprising numerous repositories.

DMBSim simulation environment is the fourth contribution of this thesis; it provides a simulation environment implanting the sequential index, inverted index, DM-index model, Chord algorithm and DNEB-Chord algorithm. The simulator is designed by C# with Visual Studio, which can validate the proposed efficient indexing and searching model for service discovery in the decentralised environment in numerous simulation scenarios.

6.2 Future Research Directions

The DM-index and DNEB-Chord model has been presented and evaluated as an efficient approach for service discovery in decentralised systems comprising large number of repositories with massive stored service. Thus the research has reached the proposed aim and objectives, but there are still areas of improvements. The next discussion details the recommended work to improve the quality of this research.

In the proposed DM-index model in Chapter 3, the effects of redundancy have been eliminated to narrow the searching space and reduce traversed services. Still, one of the future works of this research is to explore the optimal input parameter selection instead of randomly selection for service addition operation. Meanwhile, balancing the load among the distributed repositories in decentralised systems and identifying the threshold for DM-index adaptive deployment are other critical issues for future research.

The proposed approach of optimisation searching algorithm presented in Chapter 4 performs better routing mechanism and routing index than Chord with less maintenance burden, in such a way that the destination repository could be located with better efficiency. Determinate the optimal index entries for super and normal routing index to achieve better efficiency is still one tough

issue of the future works. As the infrastructure of each repository should be considered for upper-layer routing mechanism management, improving the ability of the proposed model to deal with more complex situations is another open issue for future research.

To this end, deploying and evaluating the proposed approach in a more complex and dynamic real decentralised environment is the most important challenge for future research.

References

[1] K. Holley and E. M. Tuggle, ‘Migrating to a service-oriented architecture’, no. April, 2004.

[2] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, ‘Service-oriented computing: State of the art and research challenges’, *Computer (Long. Beach. Calif).*, vol. 40, no. 11, pp. 38–45, 2007.

[3] P. Lago, T. Jansen, and M. Jansen, ‘The Service Greenery: Integrating Sustainability In Service Oriented Software’, *{ICSE} Companion - Second Int. Work. Softw. Res. Clim. Chang.*, no. June 2015, pp. 1–2, 2010.

[4] M. P. Papazoglou and W. J. Van Den Heuvel, ‘Service oriented architectures: Approaches, technologies and research issues’, *VLDB J.*, vol. 16, no. 3, pp. 389–415, 2007.

[5] ‘Introduction to Web Services Technologies: SOA, SOAP, WSDL and UDDI | Web services and the service-oriented architecture (SOA) | InformIT’. [Online]. Available: http://www.informit.com/articles/article.aspx?p=336265. [Accessed: 01-Oct-2017].

[6] H. Kreger, ‘Fulfilling the Web Services Promise’, *Commun. ACM*, vol. 46, no. 6, p. 29 ff, 2003.

[7] J. Yang, ‘Web service componentization’, *Commun. ACM*, vol. 46, no. 10, p. 35, 2003.

[8] A. Dhesiaseelan and V. Ragunathan, ‘Web services container reference architecture (WSCRA)’, *Proc. - IEEE Int. Conf. Web Serv.*, pp. 806–807, 2004.

[9] M. H. Syed, E. B. Fernandez, and M. Ilyas, ‘A Pattern for Fog Computing’, *Proc. 10th Travel. Conf. Pattern Lang. Programs - VikingPLoP ’16*, pp. 1–10, 2016.

[10] ‘Gartner Says 6.4 Billion Connected "Things" Will Be in Use in 2016, Up 30 Percent From 2015’. [Online]. Available: http://www.gartner.com/newsroom/id/3165317. [Accessed: 01-Oct-2017].

[11] U. S. Profile, ‘T H E D I G I T AL U N I V E R S E I N 2 0 2 0 : B i g D a t a , Bigger Digital Shadows , and Biggest Growth in the Far East — United States’, pp. 1–7, 2013.

[12] ‘Cisco Visual Networking Index: Forecast and Methodology, 2016–2021 - Cisco’. [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visu

al-networking-index-vni/complete-white-paper-c11-481360.html. [Accessed: 01-Oct-2017].

[13] M. Hilbert and P. López, 'The World ' s Technological Capacity to Store , Communicate , and Compute Information', 2008.

[14] A. Tzanakaki, M. P. Anastasopoulos, S. Peng, B. Rofoee, Y. Y. D. Simeonidou, G. Landi, G. Bernini, N. Ciulli, J. F. Riera, E. Escalona, K. Katsalis, and T. Korakis, 'A Converged Network Architecture for Energy Efficient Mobile Cloud Computing', pp. 19–22, 2014.

[15] D. C. Chou, 'Cloud computing: A value creation model', *Comput. Stand. Interfaces*, vol. 38, pp. 72–77, 2015.

[16] B. Wang, Z. Qi, R. Ma, H. Guan, and A. V. Vasilakos, 'A survey on data center networking for cloud computing', *Comput. Networks*, vol. 91, pp. 528–547, 2015.

[17] A. Botta, W. De Donato, V. Persico, and A. Pescapé, 'Integration of Cloud computing and Internet of Things: A survey', *Futur. Gener. Comput. Syst.*, vol. 56, pp. 684–700, 2016.

[18] R. Deng, R. Lu, C. Lai, and T. H. Luan, 'Towards power consumption-delay tradeoff by workload allocation in cloud-fog computing', *IEEE Int. Conf. Commun.*, vol. 2015–Septe, pp. 3909–3914, 2015.

[19] P. Zhang, Z. Chen, J. K. Liu, K. Liang, and H. Liu, 'An efficient access control scheme with outsourcing capability and attribute update for fog computing', *Futur. Gener. Comput. Syst.*, 2016.

[20] K. Choo, 'Cloud computing: challenges and future directions', *Trends Issues Crime Crim. Justice*, no. 400, pp. 1–6, 2010.

[21] S. Misra and S. Sarkar, 'Theoretical modelling of fog computing: a green computing paradigm to support IoT applications', *IET Networks*, vol. 5, no. 2, pp. 23–29, 2016.

[22] J. Preden, J. Kaugerand, E. Suurjaak, S. Astapov, L. Motus, and R. Pahtma, 'Data to decision: Pushing situational information needs to the edge of the network', *2015 IEEE Int. Multi-Disciplinary Conf. Cogn. Methods Situat. Aware. Decis. CogSIMA 2015*, pp. 158–164, 2015.

[23] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, 'Fog computing and its role in the internet of things', *Proc. first Ed. MCC Work. Mob. cloud Comput. - MCC '12*, p. 13, 2012.

[24] A. Al-fuqaha, S. Member, M. Guizani, M. Mohammadi, and S. Member, 'Internet of Things : A Survey on Enabling', vol. 17, no. 4, pp. 2347–2376, 2015.

[25] L. Zhang, Z. Wang, T. Mei, and D. D. Feng, 'A Scalable Approach for Content-Based Image Retrieval in Peer-to-Peer Networks', *Tkde*, vol. 28, no. 4, pp. 858–872, 2016.

[26] N. Kumar and J. H. Lee, ‘Peer-to-peer cooperative caching for data dissemination in urban vehicular communications’, *IEEE Syst. J.*, vol. 8, no. 4, pp. 1136–1144, 2014.

[27] Z. Yin, S. Member, C. Wu, and Z. Yang, ‘Peer-to-Peer Indoor Navigation Using Smartphones’, vol. 35, no. 5, pp. 1141–1153, 2017.

[28] S. Saroiu, P. K. Gummadi, and S. D. Gribble, ‘A measurement study of peer-to-peer file sharing systems’, *Proc. Multimed. Comput. Netw.*, vol. 2002, p. 152, 2002.

[29] J. Salter, ‘An Efficient Reactive Model for Resource Discovery in DHT-Based Peer-to-Peer Networks’, no. October, 2006.

[30] J. Salter and N. Antonopoulos, ‘Optimising P2P Networks by Adaptive Overlay Construction’, pp. 882–884, 2009.

[31] Y. Wu, C. G. Yan, L. Liu, Z. J. Ding, and C. J. Jiang, ‘An Adaptive Multilevel Indexing Method for Disaster Service Discovery’, *IEEE Trans. Comput.*, vol. 64, no. 9, pp. 2447–2459, 2015.

[32] Y. Wu, C. Yan, Z. Ding, G. Liu, P. Wang, C. Jiang, and M. C. Zhou, ‘A Multilevel Index Model to Expedite Web Service Discovery and Composition in Large-Scale Service Repositories’, *IEEE Trans. Serv. Comput.*, vol. 9, no. 3, pp. 330–342, 2016.

[33] I. Stoica, I. Stoica, R. Morris, D. Karger, R. Morris, D. Karger, M. F. Kaashoek, H. Balakrishnan, M. F. Kaashoek, and H. Balakrishnan, ‘Chord: A scalable peer-to-peer lookup service for internet applications’, *Proc. 2001 Conf. Appl. Technol. Archit. Protoc. Comput. Commun. - SIGCOMM ’01*, pp. 149–160, 2001.

[34] S. Ratnasamy, P. Francis, M. Handley, S. Shenker, and R. Karp, ‘A Scalable Content-Addressable Network’.

[35] A. Rowstron and P. Druschel, ‘Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems’, *Middlew. 2001*, vol. 2218, no. November 2001, pp. 329–350, 2001.

[36] M. Aazam and E. N. Huh, ‘Dynamic resource provisioning through Fog micro datacenter’, *2015 IEEE Int. Conf. Pervasive Comput. Commun. Work. PerCom Work. 2015*, pp. 105–110, 2015.

[37] M. Aazam and E. N. Huh, ‘Fog computing micro datacenter based dynamic resource estimation and pricing model for IoT’, *Proc. - Int. Conf. Adv. Inf. Netw. Appl. AINA*, vol. 2015–April, pp. 687–694, 2015.

[38] C. T. Do, N. H. Tran, C. Pham, M. G. R. Alam, J. H. Son, and C. S. Hong, ‘A proximal algorithm for joint resource allocation and minimizing carbon footprint in geo-distributed fog computing’, *Int. Conf. Inf. Netw.*, vol. 2015–Janua, pp. 324–329, 2015.

[39] ‘SOLUTION: Service Oriented Architecture - Studypool’. [Online]. Available:

https://www.studypool.com/services/243685/service-oriented-architecture. [Accessed: 01-Oct-2017].

[40] 'WindowsDevCenter.com'. [Online]. Available: http://www.windowsdevcenter.com/pub/a/dotnet/2003/08/18/soa_explained.html. [Accessed: 01-Oct-2017].

[41] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. F. Ferguson, *Web services platform architecture : SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and more*. Prentice Hall PTR, 2005.

[42] D. Krafzig, K. Banke, and D. Slama, *Enterprise SOA : service-oriented architecture best practices*. Prentice Hall Professional Technical Reference, 2005.

[43] A. Kassim, B. Esfandiari, S. Majumdar, and L. Serghi, 'A Flexible Hybrid Architecture for Management of Distributed Web Service Registries Department of Systems and Computer Engineering', pp. 2–4, 2007.

[44] M. Rambold, H. Kasinger, F. Lautenbacher, and B. Bauer, 'Towards autonomic service discovery - A survey and comparison', *SCC 2009 - 2009 IEEE Int. Conf. Serv. Comput.*, no. Section II, pp. 192–201, 2009.

[45] C. Schmidt and M. Parashar, 'A Peer-to-Peer Approach to Web Service Discovery', *World Wide Web Internet Web Inf. Syst.*, vol. 7, no. 2, pp. 211–229, 2004.

[46] R. Li, Z. Zhang, W. Song, F. Ke, and Z. Lu, 'Service Publishing and Discovering Model in a Web Services Oriented Peer-to-Peer System * 2 Web Services Oriented Peer-to-peer (WSOP) Architecture 3 Service Publishing and Discovery Model Based on WSOP Architecture', *New York*, pp. 597–599, 2005.

[47] Z. Du, J. Huai, and Y. Liu, 'Ad-UDDI: An active and distributed service registry', *Technol. E-Services*, pp. 58–71, 2006.

[48] S. Li, C. Xu, Z. Wu, Y. Pan, and X. Li, 'ABSDM : Agent Based Service Discovery Mechanism 2 The Kernel Algorithms of ABSDM', no. 602045, pp. 441–444, 2004.

[49] S. Ran, 'A model for web services discovery with QoS', *SIGecom Exch.*, vol. 4, no. 1, pp. 1–10, 2003.

[50] H. Xianglan, L. Yangguang, and X. Bin, 'Selection', 2011.

[51] S. Lamparter, A. Ankolekar, R. Studer, and S. Grimm, 'Preference-based selection of highly configurable web services', *Proc. 16th Int. Conf. World Wide Web - WWW '07*, p. 1013, 2007.

[52] L. Zeng, B. Benatallah, A. H. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang, 'QoS-aware middleware for Web services composition', *IEEE Trans. Softw. Eng.*, vol. 30, no. 5, pp. 311–327, 2004.

[53] M. Alrifai and T. Risse, ‘Combining global optimization with local selection for efficient QoS-aware service composition’, *Proc. 18th Int. Conf. World wide web - WWW ’09*, p. 881, 2009.

[54] S. Narayanan and S. a. McIlraith, ‘Simulation, verification and automated composition of web services’, *Proc. Elev. Int. Conf. World Wide Web - WWW ’02*, pp. 77–88, 2002.

[55] F. Moller and S. a. Smolka, ‘On the computational complexity of bisimulation’, *ACM Comput. Surv.*, vol. 27, no. 2, pp. 287–289, 1995.

[56] X. Tang, C. Jiang, and M. Zhou, ‘Automatic Web service composition based on Horn clauses and Petri nets’, *Expert Syst. Appl.*, vol. 38, no. 10, pp. 13024–13031, 2011.

[57] C. S. Wu and I. Khoury, ‘Tree-based search algorithm for web service composition in SaaS’, *Proc. 9th Int. Conf. Inf. Technol. ITNG 2012*, pp. 132–138, 2012.

[58] I. Constantinescu, B. Faltings, and W. Binder, ‘Large scale, type-compatible service composition’, *Proc. - IEEE Int. Conf. Web Serv.*, pp. 506–513, 2004.

[59] J. Kwon, H. Kim, D. Lee, and S. Lee, ‘Redundant-free web services composition based on a two-phase algorithm’, *Proc. IEEE Int. Conf. Web Serv. ICWS 2008*, pp. 361–368, 2008.

[60] D. Lee, J. Kwon, S. Lee, S. Park, and B. Hong, ‘Scalable and efficient web services composition based on a relational database’, *J. Syst. Softw.*, vol. 84, no. 12, pp. 2139–2155, 2011.

[61] J. Xu, W. Yu, K. Chen, and S. Reiff-Marganiec, ‘Web services feature interaction detection based on situation calculus’, *Proc. - 2010 6th World Congr. Serv. Serv. 2010*, pp. 213–220, 2010.

[62] B. Medjahed, a Bouguettaya, and a Elmagarmid, ‘{C}omposing {W}eb {S}ervices on the {S}emantic{W}eb’, *VLDB J.*, vol. 12, no. 4, 2003.

[63] R. Hamadi and B. Benatallah, ‘A Petri net-based model for web service composition’, *ADC ’03 Proc. fourteenth Australas. database Conf.*, pp. 191–200, 2003.

[64] G. J. Liu, C. J. Jiang, M. C. Zhou, and P. C. Xiong, ‘Interactive Petri nets’, *IEEE Trans. Syst. Man, Cybern. Part ASystems Humans*, vol. 43, no. 2, pp. 291–302, 2013.

[65] The_Centre_for_Energy-Efficient_Telecommunications, ‘The Power of Wireless Cloud’, *Whitepaper*, 2013.

[66] X. Ma, Y. Cui, and I. Stojmenovic, ‘Energy E ffi ciency on Location Based Applications in Mobile Cloud Computing : A Survey’, vol. 10, pp. 577–584, 2012.

[67] R. Yu, Y. He, H. Talebian, N. S. Safa, N. Zhao, M. K. Khan, and N. Kumar, ‘Fog Vehicular Computing’, no. September, pp. 2–11, 2017.

[68] and J. M. K. Verma, K. Sivashanmugam, A. Sheth, A. Patil, S. Oundhakar, ‘N141 \’, *Inf. Technol. Manag.*, vol. 6, pp. 17–39, 2005.

[69] P. a. Grabowicz, J. J. Ramasco, E. Moro, J. M. Pujol, and V. M. Eguiluz, ‘Social features of online networks: The strength of intermediary ties in online social media’, *PLoS One*, vol. 7, no. 1, pp. 1–9, 2012.

[70] K. Verma, K. Sivashanmugam, A. Sheth, A. Patil, S. Oundhakar, and J. Miller, ‘METEOR-S WSDI: A scalable P2P infrastructure of registries for semantic publication and discovery of web services’, *Inf. Technol. Manag.*, vol. 6, no. 1, pp. 17–39, 2005.

[71] IEEE Computer Society., Association for Computing Machinery., and Sigarch., *SC2004 : bridging communities, November 6-12, 2004*. IEEE Computer Society, 2004.

[72] E. Nikolaos, C. Athanasios, D. Spiros, and K. Odysseas, ‘Enabling locality in a balanced peer-to-peer overlay’, 2006.

[73] V. Vishnevsky, A. Safonov, M. Yakimov, E. Shim, and A. D. Gelman, ‘Tag Routing for Efficient Blind Search in Peer-to-Peer Networks’, pp. 0–7, 2006.

[74] S. Sivaraja, M. Thiyagarajah, and T. Piranavam, ‘Efficient Multiple-Keyword Search in DHT-based Decentralized Systems’, 2008.

[75] S. Khorsandi, ‘Popularity-Based Globally Structured Hybrid Peer-to-Peer Network’, pp. 411–415, 2010.

[76] C. Huang, M. Ma, Y. Liu, and A. Liu, ‘Preserving Source Location Privacy for Energy Harvesting WSNs’, *Sensors*, vol. 17, no. 4, p. 724, 2017.

[77] J. He, J. Wei, K. Chen, Z. Tang, Y. Zhou, and Y. Zhang, ‘Multi-tier Fog Computing with Large-scale IoT Data Analytics for Smart Cities’, *IEEE Internet Things J.*, vol. 4662, no. c, pp. 1–10, 2017.

[78] C. Cover, ‘Fog Computing : Helping the Internet of Things Realize’, 2016.

[79] M. T. Saqib and M. A. Hamid, ‘FogR: A highly reliable and intelligent computation offloading on the Internet of Things’, *IEEE Reg. 10 Annu. Int. Conf. Proceedings/TENCON*, pp. 1039–1042, 2017.

[80] L. M. Vaquero and L. Rodero-Merino, ‘Finding your Way in the Fog: Towards a Comprehensive Definition of Fog Computing’, *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 5, pp. 27–32, 2014.

[81] D. Sun and X. Wang, ‘Inverted Index Compression using Multi-codes’, 2014.

[82] H. Garcia-Molina, J. D. Ullman, J. Widom, M. Özsu, P. Valduriez, T. Connolly, C. Begg, R. Elmasri, S. B. Navathe, M. Lin, M. Tsuchiya, S. Member, M. P. Mariani, M. Sharma, G. Singh, and R. Virk, *Database Systems: A Practical Approach to Design, Implementation, and Management*, vol. 49, no. 4. 2010.

[83] T.-W. Kuo, C.-H. Wei, and K.-Y. Lam, 'Real-Time Data Access Control on B-Tree Index Structures', *IEEE Int. Conf. Data Eng.*, pp. 458–467, 1999.

[84] W. Jung, H. Roh, M. Shin, and S. Park, 'Inverted index maintenance strategy for flashSSDs : Revitalization of in-place index update strategy', *Inf. Syst.*, vol. 49, pp. 25–39, 2015.

[85] W. Jung, H. Roh, M. Shin, and S. Park, 'Inverted index maintenance strategy for flashSSDs: Revitalization of in-place index update strategy', *Inf. Syst.*, vol. 49, pp. 25–39, Apr. 2015.

[86] L. Aversano, G. Canfora, A. Ciampi, and V. Traiano, 'An algorithm for Web service discovery through their composition Department of Engineering , University of Sannio', pp. 1–8.

[87] K. Li, L. Ying, W. Jian, D. Shuiguang, and W. Zhaohui, 'Inverted indexing for composition-oriented service discovery', *Proc. - 2007 IEEE Int. Conf. Web Serv. ICWS 2007*, no. 60603025, pp. 257–264, 2007.

[88] S. Shah, 'Inverted Index'.

[89] K. Santhiya, 'Map Reduce Programming Model : Construction of Inverted Index for Automated Document Clustering', pp. 308–312, 2016.

[90] J. Giridharan and S. V Vairavan, 'Inverted Index and Interval Lists for Keyword Search', pp. 2–5.

[91] H. Wu, G. Li, and L. Zhou, 'Ginix : Generalized Inverted Index for Keyword Search', vol. 18, no. 1, 2013.

[92] Q. He, J. Yan, Y. Yang, R. Kowalczyk, H. Jin, and S. Member, 'A Decentralized Service Discovery Approach on Peer-to-Peer Networks', vol. 6, no. 1, pp. 64–75, 2013.

[93] Y. H. Chen, E. J. L. Lu, Y. T. Chang, and S. Y. Huang, 'RDF-Chord: A hybrid PDMS for P2P systems', *Comput. Stand. Interfaces*, vol. 43, pp. 53–67, 2016.

[94] H. Wirtz, T. Heer, M. Serror, and K. Wehrle, 'DHT-based localized service discovery in wireless mesh networks', *MASS 2012 - 9th IEEE Int. Conf. Mob. Ad-Hoc Sens. Syst.*, pp. 19–28, 2012.

[95] S. Cirani and L. Veltri, 'Implementation of a framework for a DHT-based distributed location service', *SoftCom 2008 16th Int. Conf. Software, Telecommuncations Comput. Networks*, pp. 279–283, 2008.

[96] S. Ratnasamy, P. Francis, M. Handley, S. Shenker, and R. Karp, ‘A Scalable Content-Addressable Network’.

[97] P. Maymounkov and D. Mazieres, ‘Kademlia: A peer-to-peer information system based on the xor metric’, *First Int. Work. Peer-to-Peer Syst.*, pp. 53–65, 2002.

[98] a.-T. Gai and L. Viennot, ‘Broose: a practical distributed hashtable based on the de-Bruijn topology’, *Proceedings. Fourth Int. Conf. Peer-to-Peer Comput. 2004. Proceedings.*, 2004.

[99] H. S. Nguyen, H. Bui, T. Lan, and N. A. Nguyen, ‘Topology Optimization for DHT-based Application Layer Multicast’, 2012.

[100] A. Macquire, I. A. Rai, and N. J. P. Race, ‘Authentication in Stealth Distributed Hash Tables’, 2006.

[101] H. Tanta-ngai and M. McAllister, ‘A peer-to-peer expressway over Chord’, *Math. Comput. Model.*, vol. 44, no. 7–8, pp. 659–677, 2006.

[102] A. MacQuire, A. Brampton, I. A. Rai, N. J. P. Race, and L. Mathy, ‘Authentication in stealth distributed hash tables’, *J. Syst. Archit.*, vol. 54, no. 6, pp. 607–618, 2008.

[103] E. Meshkova, J. Riihij??rvi, M. Petrova, and P. M??h??nen, ‘A survey on resource discovery mechanisms, peer-to-peer and service discovery frameworks’, *Comput. Networks*, vol. 52, no. 11, pp. 2097–2128, 2008.

[104] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, ‘Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications’, *{IEEE} Trans. Netw.*, vol. 11, pp. 1–14, 2003.

[105] A. Rowstron and P. Druschel, ‘Pastry : Scalable , decentralized object location and routing for large-scale peer-to-peer systems’, no. November 2001, 1892.

[106] B. Y. Zhao, J. Kubiatowicz, A. D. Joseph, B. Y. Zhao, J. Kubiatowicz, and A. D. Joseph, ‘Tapestry : An Infrastructure for Fault-tolerant Wide-area Location and Routing Tapestry : An Infrastructure for Fault-tolerant Wide-area Location and Routing’, no. April, 2001.

[107] M. F. Kaashoek and D. R. Karger, ‘Koorde : A simple degree-optimal distributed hash table’, no. 1.

[108] P. Maymounkov and D. Mazi, ‘Kademlia : A Peer-to-peer Information System Based on the XOR Metric’.

[109] J. Moorhouse, Lu Liu, Zhiyuan Li, and Zhijun Ding, ‘Modelling and Simulation of Peer-to-Peer Overlay Network Protocols using OverSim’, in *2013 UKSim 15th International Conference on Computer Modelling and Simulation*, 2013, pp. 144–149.

[110] H. Nagao and K. Shudo, ‘Flexible routing tables: Designing routing algorithms for overlays based on a total order on a routing table set’, *2011 IEEE Int. Conf. Peer-to-Peer Comput. P2P 2011 - Proc.*, pp. 72–81, 2011.

[111] E. J. L. Lu, Y. F. Huang, and S. C. Lu, ‘ML-Chord: A multi-layered P2P resource sharing model’, *J. Netw. Comput. Appl.*, vol. 32, no. 3, pp. 578–588, 2009.

[112] R. Morris, J. Li, and others, ‘Routing tradeoffs in dynamic peer-to-peer networks’, *Work*, no. 1998, 2005.

[113] L. Zhao, H. Shen, Y. Li, and J. Wu, ‘AB-Chord : An Improved Chord Based on Ant Colony Optimizationand Bi-directional Lookup Routing’, 2014.

[114] H. Zhang, X. Du, C. Zhang, and A. Chord, ‘Improvement of Structured P2P Routing Algorithm Based on NN-Chord’, 2011.

[115] Z. X. L. I. U. Fang-ai, ‘MF-Chord : Supporting Multi-Attribute Multi-keyword Fuzzy-Matching Queries’, 2009.

[116] C. Jeyabharathi, ‘Improvement of Chord Algorithm in Efficient Grid Resource Discovery : A Comparative Analysis’, 2013.

[117] G. Chiola and M. Ribaudo, ‘Neighbor-of-neighbor routing over deterministically modulated Chord-like DHTs’, *IPDPS Miami 2008 - Proc. 22nd IEEE Int. Parallel Distrib. Process. Symp. Progr. CD-ROM*, 2008.

[118] B. Dai, F. Wang, J. Ma, and J. Liu, ‘Enhanced chord-based routing protocol using neighbors’ neighbors links’, *Proc. - Int. Conf. Adv. Inf. Netw. Appl. AINA*, no. 2006, pp. 463–466, 2008.

[119] C. Cheng, Y. Xu, and X. Xu, ‘Advanced chord routing algorithm based on redundant information replaced and objective resource table’, *Proc. - 2010 3rd IEEE Int. Conf. Comput. Sci. Inf. Technol. ICCSIT 2010*, vol. 6, pp. 247–250, 2010.

[120] W. Lv, Q. Liao, J. Zhao, and Y. Xiao, ‘TB _ Chord : An Improved Routing Algorithm to Chord Based on Topology-aware and Bi-Dimensional Lookup Method’, pp. 4–7, 2009.

[121] D. Chen, Z. Tan, G. Chang, and X. Wang, ‘An Improvement to the Chord-based P2P Routing Algorithm’, vol. 63, pp. 6–9, 2009.

[122] D. Chen, Z. Tan, G. Chang, and X. Wang, ‘An improvement to the chord-based P2P routing algorithm’, *SKG 2009 - 5th Int. Conf. Semant. Knowledge, Grid*, vol. 63, pp. 266–269, 2009.

[123] M. Dorigo, S. Member, and L. M. Gambardella, ‘Ant Colony System : A Cooperative Learning Approach to the Traveling Salesman Problem’, vol. 1, no. 1, pp. 53–66, 1997.

[124] L. Ye, Y. Peng, C. Zi, and W. Jiagao, ‘TCS-Chord : An Improved Routing Algorithm to Chord Based on the Topology-aware Clustering in Self-organizing Mode’, no. Skg 2005, 2006.

[125] J. Salter, ‘An Efficient Reactive Model for Resource Discovery in DHT-Based Peer-to-Peer Networks’, no. October, 2006.

[126] Z. Guo, L. Min, S. Yang, and H. Yang, ‘An enhanced P4P-based Pastry routing algorithm for P2P network’, *Proc. - 2010 IEEE Int. Conf. Granul. Comput. GrC 2010*, no. 3, pp. 687–691, 2010.

[127] D. Battr??, A. H??ing, M. Raack, U. Rerrer-Brusch, and O. Kao, ‘Extending pastry by an alphanumerical overlay’, *2009 9th IEEE/ACM Int. Symp. Clust. Comput. Grid, CCGRID 2009*, pp. 36–43, 2009.

[128] J. Furness, M. Kolberg, and M. Fayed, ‘An evaluation of chord and pastry models in OverSim’, *Proc. - UKSim-AMSS 7th Eur. Model. Symp. Comput. Model. Simulation, EMS 2013*, pp. 509–513, 2013.

[129] D. Nishimura, ‘Graphically speaking’, *Science (80-.).*, vol. 283, no. 5405, p. 1134, 1999.

[130] X. Lian, A. Fakhry, L. Zhang, and J. Cleland-Huang, ‘Leveraging Traceability to Reveal the Tapestry of Quality Concerns in Source Code’, *Proc. - 2015 IEEE/ACM 8th Int. Symp. Softw. Syst. Traceability, SST 2015*, pp. 50–56, 2015.

[131] H. Heck, O. Kieselmann, and A. Wacker, ‘Evaluating Connection Resilience for the Overlay Network Kademlia’, *2017 IEEE 37th Int. Conf. Distrib. Comput. Syst.*, pp. 2581–2584, 2017.

[132] L. Zhu and K. Zheng, ‘An improved Kademlia algorithm based on Qos’, *Proc. 2014 Int. Conf. Cloud Comput. Internet Things, CCIOT 2014*, no. Cctot, pp. 128–130, 2014.

[133] L. Chen, W. S. Soh, and A. Hu, ‘An improved Kademlia protocol with double-layer design for P2P voice communications’, *Commun. Secur. Conf. (CSC 2014), 2014*, pp. 1–8, 2014.

[134] M. Wei and G. Dong, ‘Improvement of Kademlia based on physical location’, *Proc. - 2013 10th Web Inf. Syst. Appl. Conf. WISA 2013*, pp. 119–122, 2013.

[135] Z. Wang, Z. Wang, and J. Crowcroft, ‘Analysis of Shortest-Path Routing Algorithms in a Dynamic Network Environment’, *ACM Comput. Commun. Rev.*, vol. 22, pp. 63--71, 1992.

[136] A. Montresor, ‘PeerSim : A Scalable P2P Simulator ∗’, no. 214412, pp. 99–100, 2009.

[137] I. Baumgart, B. Heep, and S. Krause, ‘OverSim : A Flexible Overlay Network Simulation Framework’, pp. 79–84, 2007.

[138] T. M. Gil, M. F. Kaashoek, J. Li, R. Morris, and J. Stribling, ‘p2psim : a simulator for peer-to-peer protocols 1 Introduction’.

[139] J. Pujol-ahull, ‘PlanetSim : An extensible simulation tool for peer-to-peer networks and services’, pp. 85–86, 2009.

[140] I. Kazmi and S. F. Y. Bukhari, ‘PeerSim: An efficient scalable testbed for heterogeneous cluster-based P2P network protocols’, *Proc. - 2011 UKSim 13th Int. Conf. Model. Simulation, UKSim 2011*, pp. 420–425, 2011.

[141] I. Kazmi, ‘PeerSim : An Efficient & Scalable Testbed for Heterogeneous Cluster-based P2P Network Protocols’, 2011.

[142] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker, ‘Making Gnutella-like P2P Systems Scalable’, 2003.

[143] I. Baumgart, B. Heep, and S. Krause, ‘OverSim : A Flexible Overlay Network Simulation Framework’, pp. 79–84, 2007.

[144] I. Baumgart, B. Heep, and S. Krause, ‘OverSim: A Flexible Overlay Network Simulation Framework’, *2007 IEEE Glob. Internet Symp.*, pp. 79–84, 2007.

[145] S. Surati, D. C. Jinwala, and S. Garg, ‘Engineering Science and Technology , an International Journal A survey of simulators for P2P overlay networks with a case study of the P2P tree overlay using an event-driven simulator’, vol. 20, pp. 705–720, 2017.

[146] S. Engineering and I. Technologies, ‘AUTOMATIC DYNAMIC WEB SERVICE COMPOSITION : A SURVEY AND PROBLEM FORMALIZATION Peter Bartalos , M ´aria Bielikov a’, *Comput. Informatics*, vol. 30, pp. 793–827, 2011.

[147] ‘Kolman, Busby & Ross, Discrete Mathematical Structures, 6th Edition’. [Online]. Available: https://www.pearson.com/us/higher-education/program/Kolman-Discrete-Mathematical-Structures-6th-Edition/PGM104913.html. [Accessed: 03-Oct-2017].

[148] K. H. Rosen, *Discrete mathematics and its applications*, vol. 10. 2014.

[149] ‘Bijection, injection and surjection - Wikipedia’. [Online]. Available: https://en.wikipedia.org/wiki/Bijection,_injection_and_surjection. [Accessed: 04-Oct-2017].

[150] K. E. S. Desikan, M. Srinivasan, and C. S. R. Murthy, ‘A Novel Distributed Latency-Aware Data Processing in Fog Computing-Enabled IoT Networks’, *Proc. ACM Work. Distrib. Inf. Process. Wirel. Networks - DIPWN’17*, pp. 1–6, 2017.

[151] S. Cirani and L. Veltri, ‘Implementation of a framework for a DHT-based Distributed Location Service’.

[152] M. Swathi, B. Pravallika, and N. V Muralidhar, ‘Implementing And Comparison of MANET Routing Protocols Using NS2’, vol. 4, no. 2, pp. 194–200, 2014.

[153] Y. Wang, Q. Jin, X. Li, and J. Ma, ‘AB-chord: An efficient approach for resource location in structured P2P networks’, *Proc. - IEEE 9th Int. Conf. Ubiquitous Intell. Comput. IEEE 9th Int. Conf. Auton. Trust. Comput. UIC-ATC 2012*, pp. 278–284, 2012.

[154] Y. Wang, Q. Jin, X. Li, and J. Ma, ‘AB-chord: An efficient approach for resource location in structured P2P networks’, *Proc. - IEEE 9th Int. Conf. Ubiquitous Intell. Comput. IEEE 9th Int. Conf. Auton. Trust. Comput. UIC-ATC 2012*, pp. 278–284, 2012.

[155] A. Gupta, B. Liskov, and R. Rodrigues, ‘One Hop Lookups for Peer-to-Peer Overlays’, *9th Work. Hot Top. Oper. Syst.*, pp. 1–6, 2003.

[156] D. Loguinov, J. Casas, and X. Wang, ‘Graph-theoretic analysis of structured peer-to-peer systems: Routing distances and fault resilience’, *IEEE/ACM Trans. Netw.*, vol. 13, no. 5, pp. 1107–1120, 2005.

[157] J. Moorhouse, L. Liu, Z. Li, and Z. Ding, ‘Modelling and Simulation of Peer-to-Peer Overlay Network Protocols Using OverSim’, 2013.

Appendix

A1 Acronyms

No	Acronysms	Meaning
1	DM-index	Distributed Multilevel Index
2	DHTs	Distributed Hash Tables
3	DNEB-Chord	Double-layer No-redundancy Enhanced Bi-direction Chord
4	DMBSim	Distributed Multilevel Bi-direction Simoulator
5	ICT	Information and Communication Technology
6	CC	Clouding Computing
7	FC	Fog Computing
8	SOA	Service-Oriented Architecture
9	SOC	Service-Oriented Computing
10	P2P	Peer-to-Peer
11	IOT	Internet of Things
12	SOAP	Simple Object Access Protocol
13	WSDL	Web Services Description Language
14	WSOP	Web Services Oriented Peer-to-Peer
15	CWS	Common Web Service
16	LSRB	Local Service Registry Broker
17	CRSB	Common Service Registry Broker
18	ABSDM	Agent-based Service Discovery Mrchanim
19	QoS	Quanlity of Service
20	DAML—S	DARPA agent markup language for service
21	SaaS	Soft as a Service
22	PaaS	Platform as a Service
23	IaaS	Infrastruate as a Service
24	NIST	US Department of Commerce National Institute of Standard and Technology

A2 Notations

No	Notation	Meaning
1	s	service
2	P_{si}	input parameters set of service
3	P_{so}	output parameters set of service
4	$\mathrm{P_{sr}}$	remaining parameters
5	Q	User request
6	Q_p	the parameters set provided by user
7	Q_r	the parameters set required by user
8	S	services set
9	L	a set of constraints for any attributes
10	R	relation
11	C_s	same-class set
12	P_{csi}	input parameters set of same-class
13	P_{cso}	output parameters set of same-class
14	C_{is}	semi-class set
15	P_{csi}	input parameters set of semi-class
16	P_{cso}	output parameters set of semi-class
17	P_{rs}	all the randomly selected input parameters set
18	I_r	the randomly selected input parameters list
19	E_s	the expectation of the traversed services for the sequential index
20	E_i	the expectation of the traversed services for the inverted index
21	P	the input parameter pool of all stored services and retrieval requests
22	P_i	the average number of input parameters of each stored service
23	E_{fl}	the expectation of the traversed services for the full DM-index model
24	E_{pr}	the expectation of the traversed services for the primary deployment model of DM-index
25	E_{pt}	the expectation of the traversed services for the partial deployment model of DM-index

A3: C# Code for DMBSim

A3.1 Common Setting

```
namespace multilevelindex
{
    class Common
    {
        private static bool _flag = false;
        private static int _size = 4;
        private static bool _debug = false;
        private static int _retrievalNum = 0;
        private static int _lines = 12;
        private static bool _ifCounter = false;
        private static int _idSpace = 16;
        private static int _retriveNum = 0;
        private static int _nodeCounts = 0;
        private static int _nodes = 100;
        public Common()
        {
        }
        public static void setNodeNumber(int nodes)
        {
            _nodes = nodes;
        }
        public static int getNodeNumber()
        {
            return _nodes;
        }
        public static void setNodeCounts(int nodeCounts)
        {
            _nodeCounts = nodeCounts;
        }
        public static int getNodeCounts()
        {
            return _nodeCounts;
        }
        public static void setRetriveNum(int retriveNum)
        {
            _retriveNum = retriveNum;
        }
```

```
public static int getRetriveNum()
{
    return _retriveNum;
}
public static void setIDSpace(int idSpace)
{
    _idSpace = idSpace;
}
public static int getIDSpace()
{
    return _idSpace;
}
public static void SetFlag(bool flag)
{
    _flag = flag;
}
public static bool GetFlag()
{
    return _flag;
}
public static void SetSize(int size)
{
    _size = size;
}
public static int GetSize()
{
    if (_flag)
    {
        return _size;
    }
    else
    {
        return 0;
    }
}
public static void SetLine(int line)
{
    _lines = line;
}

public static int GetLine()
{
```

```
    return _lines;
}
public static void SetCounterCircle(bool ifCounter)
{
    _ifCounter = ifCounter;
}
public static bool GetCounterCircle()
{
    return _ifCounter;
}
public static void SetDebug(bool debug)
{
    _debug = debug;
}
public static bool GetDebug()
{
    return _debug;
}
public static void SetRetrievalNum(int retrievalNum)
{
    _retrievalNum += retrievalNum;
}
public static int GetRetrievalNum()
{
    return _retrievalNum;
}
public static void Clear()
{
    _retrievalNum = 0;
 }
public static bool LiesIn(int a, int x, int y)
{
    int max = 1 << _size;
    if (y < x)
    {
        if (((a > x) && (a <= max)) || ((a >= 0) && (a < y)))
            return true;
    }
    else if ((a > x) && (a < y))
        return true;
    return false;
}
```

```
        public static bool LiesInAddress(int a, Address x, Address y)
        {
            int X = x.GetMap();
            int Y = y.GetMap();
            return LiesIn(a, X, Y + 1);
        }
        public static Address InitAddress(string name, int size)
        {
            int map = FHash.MapNode(name, size);
            Address temp = new Address();

            temp.SetIpAddress(name);
            temp.SetMap(map);
            return temp;
        }
      }

}
```

A3.2 GUI of DMBSim

```
namespace multilevelindex
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            numericUpDown_servicesetsize.Value = 10000;
            numericUpDown_parametersetsize.Value = 1000;
            numericUpDown_parametercountperservice.Value = 10;
           numericUpDown_retrivalparametersetsize.Value = 10;
            numericUpDown_retrievalcount.Value = 100;
            numericUpDown_numberofnode.Value = 100;
            numericUpDown_retrievalcount.Value = 100;
            checkBox_PrimaryIndex.Checked = false;
            checkBox_mediumindex.Checked = false;
            checkBox_multilevelindex.Checked = false;
            checkBox_invertedindex.Checked = false;
            checkBox_sequenceindex.Checked = false;
            checkBox_sameseed.Checked = true;
            numericUpDown_row.Value = 11;
            checkBox_testservicesetsize.Checked = false;
            numericUpDown_initialservicesetsize.Value = 10000;
            numericUpDown_ssslength.Value = 1000;
            checkBox_testparametersetsize.Checked = false;
            numericUpDown_initialparametersetsize.Value = 1000;
            numericUpDown_Plength.Value = 100;
            checkBox_testparametercountperserivce.Checked = false;
            numericUpDown_initalnpsvalue.Value = 10;
            numericUpDown_npslength.Value = 1;
            checkBox_testretrivalparametersetsize.Checked = false;
            numericUpDown_initialnprvalue.Value = 10;
            numericUpDown_nprlength.Value = 1;
            checkBox_testnumberofnode.Checked = false;
            numericUpDown_initialnumberofnode.Value = 100;
```

```
            numericUpDown_NNlength.Value = 50;
            checkBox_testretrievalcount.Checked = false;
            numericUpDown_initialretrievalcount.Value = 100;
            numericUpDown_NRlength.Value = 50;
            textBox1.Text = Environment.CurrentDirectory + "\\results.xlsx";
        }
        private void button_exit_Click(object sender, EventArgs e)
        {
            Environment.Exit(0);
        }
        private void button_test_Click(object sender, EventArgs e)
        {
            this.Enabled = false;
            Console.WriteLine("Test Start");
            int row = (int)numericUpDown_row.Value;
            string resultfilepath = textBox1.Text;
            int seed = configuration_class.rand.Next();
            configuration_class.defaultservicesetsize =
(int)numericUpDown_servicesetsize.Value;
            configuration_class.defaultparametersetsize =
(int)numericUpDown_parametersetsize.Value;
            configuration_class.defaultparametercountperservice =
(int)numericUpDown_parametercountperservice.Value;
            configuration_class.defaultretrievalparametersetsize =
(int)numericUpDown_retrivalparametersetsize.Value;
            configuration_class.defaultretrievalcount =
(int)numericUpDown_retrievalcount.Value;
            configuration_class.defaultnumberofnode =
(int)numericUpDown_numberofnode.Value;
            configuration_class.defaultnumberoffingertable =
(int)numericUpDown_fingerTableSize.Value;
            configuration_class.defaultnumberofhash =
(int)numericUpDown_IDSpace.Value;
            Common.setNodeNumber((int)numericUpDown_numberofnode.Value);
            Common.setIDSpace((int)numericUpDown_IDSpace.Value);
            Common.SetLine((int)numericUpDown_fingerTableSize.Value);
            if (excel_class.isclosedandok(resultfilepath))
            {
                if (checkBox_PrimaryIndex.Checked)
                    testindex(configuration_class.primaryindexID, seed, row,
resultfilepath);
                if (checkBox_mediumindex.Checked)
```

```
                testindex(configuration_class.mediumindexID, seed, row,
resultfilepath);
            if (checkBox_multilevelindex.Checked)
                testindex(configuration_class.multilevelindexID, seed,
row, resultfilepath);
            if (checkBox_invertedindex.Checked)
                testindex(configuration_class.invertedindexID, seed, row,
resultfilepath);
            if (checkBox_sequenceindex.Checked)
                testindex(configuration_class.sequenceindexID, seed, row,
resultfilepath);
        }
        Console.WriteLine("Test Finished");
        this.Enabled = true;
    }
    void testindex(int testedindexID, int seed, int row, string
resultfilepath)
    {
        if (!checkBox_sameseed.Checked)
            seed = configuration_class.rand.Next();
        configuration_class.setrandomseed(seed); /
       configuration_class.setNodeRandomSeed(seed);
        if (checkBox_testservicesetsize.Checked)
        {
            testperformance_class.testperformace(testedindexID,
resultfilepath,
                configuration_class.testservicesetsize,
(int)numericUpDown_initialservicesetsize.Value,
                (int)numericUpDown_ssslength.Value, row);
        }
        if (checkBox_testparametersetsize.Checked)
        {
            testperformance_class.testperformace(testedindexID,
resultfilepath,
                configuration_class.testparametersetsize,
(int)numericUpDown_initialparametersetsize.Value,
                (int)numericUpDown_Plength.Value, row);
        }
        if (checkBox_testparametercountperserivce.Checked)
        {
            testperformance_class.testperformace(testedindexID,
resultfilepath,
```

```
                    configuration_class.testparametercountperservice,
(int)numericUpDown_initalnpsvalue.Value,
                    (int)numericUpDown_npslength.Value, row);
            }
            if (checkBox_testretrivalparametersetsize.Checked)
            {
                testperformance_class.testperformace(testedindexID,
resultfilepath,
                    configuration_class.testretrievalparametersetsize,
(int)numericUpDown_initialnprvalue.Value,
                    (int)numericUpDown_nprlength.Value, row);
            }
             if (checkBox_testnumberofnode.Checked)
             {
                testperformance_class.testperformace(testedindexID,
resultfilepath,
                    configuration_class.testnumberofnode,
(int)numericUpDown_initialnumberofnode.Value,
                    (int)numericUpDown_NNlength.Value, row);
             }
            if (checkBox_testretrievalcount.Checked)
            {
                testperformance_class.testperformace(testedindexID,
resultfilepath,
                    configuration_class.testretrievalcount,
(int)numericUpDown_initialretrievalcount.Value,
                    (int)numericUpDown_NRlength.Value, row);
            }
        }
        private void label21_Click(object sender, EventArgs e)
        {
        }
        private void label22_Click(object sender, EventArgs e)
        {
        }
        private void label18_Click(object sender, EventArgs e)
        {
        }
        private void label2_Click(object sender, EventArgs e)
        {
        }
```

```
        }
    }
```

A3.3 Configurations

```
namespace multilevelindex
{
    public static class configuration_class
    {
        public const int primaryindexID = 1;
        public const int mediumindexID = 2;
        public const int multilevelindexID = 3;
        public const int invertedindexID = 4;
        public const int sequenceindexID = 5;
        public const int testservicesetsize = 1;
        public const int testparametercountperservice = 2;
        public const int testretrievalparametersetsize = 3;
        public const int testparametersetsize = 4;

        public const int testnumberofnode = 5;
        public const int testretrievalcount = 6;
        public const string sequenceindex = "sequenceindex";
        public const string invertedindex = "invertedindex";
        public const string primaryindex = "primaryindex";
        public const string mediumindex = "mediumindex";
        public const string multilevelindex = "multilevelindex";
        public const string servicesetsize = "servicesetsize";
        public const string parametercountperservice = "parameter count per
service";
        public const string retrievalparametersetsize = "retrieval parameter
set size";
        public const string parametersetsize = "parameter set size";

        public const string numberofnode = "number of node";
        public const string retrievalcount = "retrieval count";
        public static int defaultservicesetsize = 0;
        public static int defaultparametersetsize = 0;
        public static int defaultparametercountperservice = 0;
        public static int defaultretrievalparametersetsize = 0;
        public static int defaultretrievalcount = 0;
        public static int defaultnumberofnode = 0;
        public static int defaultnumberoffingertable = 0;
        public static int defaultnumberofhash = 0;
        static int  ascendingnumber_ = 0;
```

```
        public static int ascendingnumber
        {
            get { return (ascendingnumber_++); }
        }
        static int nodeRandomSeed = 0;
        public static int getNodeRandomSeed()
        {
            return nodeRandomSeed;
        }
        public static void setNodeRandomSeed(int seed)
        {
            nodeRandomSeed = seed;
        }
        static Random rand_ = new Random();
        public static Random rand
        {
            get { return (rand_); }
        }
        public static void setrandomseed(int seed)
        {
            rand_ = new Random(seed);
        }
        public static double calculateoptimumvalue(int membercount)
        {
            double optimumvalue = Math.Sqrt(membercount);
            return (optimumvalue);
        }
        public static int[] intarray(int basenumber, int weight, int count)
        {
            int[] a = new int[count];
            for (int i = 0; i < count; i++)
            {
                a[i] = basenumber + i * weight;
            }
            return (a);
        }

    }
}
```

A3.4 Sequential Index Model

```
namespace multilevelindex
{
    class sequencedindex_class : generalindex_class
    {
        serviceset_class sequenceddindex_ = new serviceset_class();
        public sequencedindex_class()
        {
            indexID_ = configuration_class.sequenceindexID;
            indexname_ = configuration_class.sequenceindex;
        }
        public override bool add(service_class service, int dum)
        {
            return (sequenceddindex_.addtoserviceset(service));
        }
        public override bool delete(service_class service)
        {
            return (sequenceddindex_.deletefromserviceset(service));
        }
        public serviceset_class getServiceSet()
        {
            return sequenceddindex_;
        }
     }
}
```

A3.5 Inverted Index Model

```
namespace multilevelindex
{
    class invertedindex_class : generalindex_class
    {
        SortedDictionary<int, serviceset_class> invertedindex_ = new
SortedDictionary<int, serviceset_class>();

        public invertedindex_class()
        {
            indexID_ = configuration_class.invertedindexID;
            indexname_ = configuration_class.invertedindex;
        }

         public override bool add(service_class service, int selectedkey)
        {
            foreach (int inputparameter in service.inputs)
            {
                serviceset_class serviceset;
                if (invertedindex_.TryGetValue(inputparameter, out
serviceset))
                {
                    serviceset.addtoserviceset(service);
                }
                else
                {
                    serviceset = new serviceset_class();
                    serviceset.addtoserviceset(service);
                    invertedindex_.Add(inputparameter, serviceset);
                }
            }
            return (true);
        }
        public override bool delete(service_class service)
        {
            foreach (int inputparameter in service.inputs)
            {
                serviceset_class serviceset;
                if (invertedindex_.TryGetValue(inputparameter, out
serviceset))
```

```
                {
                    if (!serviceset.deletefromserviceset(service))
                        return (false);
                }
                else
                {
                    return (false);
                }
                if (serviceset.servicecount == 0)
                    invertedindex_.Remove(inputparameter);
            }
            return true;
        }
        public override LinkedList<service_class> retrieval(int
parameterset)
        {
            LinkedList<service_class> retrievedsericelist = new
LinkedList<service_class>();
            serviceset_class serviceset;
            if (invertedindex_.TryGetValue(parameterset, out serviceset))
            {
                serviceset.retrievalfromserviceset(parameterset,
retrievedsericelist);

            }

            return (retrievedsericelist);
        }
    }

}
```

A3.6 DM-Index Model _Full Deployment

```
namespace multilevelindex
{
    class multilevelindex_class : generalindex_class
    {
        SortedDictionary<int, keyclass_class> multilevelindex_ = new
SortedDictionary<int, keyclass_class>();

        public multilevelindex_class()
        {
            indexID_ = configuration_class.multilevelindexID;
            indexname_ = configuration_class.multilevelindex;
        }
        public override bool add(service_class service, int selectedkey)
        {
            semisimilarclass_class semisimilarclass = null;
            LinkedList<keyclass_class> keyclasslist = null;
            int unkey;
            if (findequalsemisimilarclass(service, out semisimilarclass, out
keyclasslist, out unkey, multilevelindex_))
            {
                if (semisimilarclass.addtosimilarclass(service))
                {
                    servicecount_++;
                    return (true);
                }
            }
            keyclass_class keyclass;
            int key = selectkey(service, keyclasslist, unkey, out keyclass,
semisimilarclasscount_);
            if (key == unkey)
            {
                multilevelindex_.Add(key, keyclass);
            }
            if (keyclass.addservicetonewsemisimilarclass(service,
configuration_class.multilevelindexID))
            {
                servicecount_++;
                semisimilarclasscount_++;
                return (true);
```

```
            }
            return (false);
        }
        public override LinkedList<service_class> retrieval(int
parameterset)
        {
            LinkedList<service_class> servicelist = new
LinkedList<service_class>();
            keyclass_class keyclass = null;

            if (multilevelindex_.TryGetValue(parameterset, out keyclass))
            {
                keyclass.retrievalfromsemisimilarclass(parameterset,
servicelist, configuration_class.multilevelindexID);
                Common.SetRetrievalNum(keyclass.semisimilarclasscount);
                return (servicelist);
            }
            else
            {
                return null;
            }
         }
        public override bool delete(service_class service)
        {
            keyclass_class keyclass = null;
            if (multilevelindex_.TryGetValue(service.key, out keyclass))
            {
                if (keyclass.deletefromsemisimilarclass(service, ref
semisimilarclasscount_, configuration_class.multilevelindexID))
                {
                    if (keyclass.semisimilarclasscount == 0)
                    {
                        multilevelindex_.Remove(service.key);
                    }
                    servicecount_--;
                    return (true);
                }
            }
            return (false);
        }
        public SortedDictionary<int, keyclass_class> getIndexDict()
        {
```

```
            return multilevelindex_;
        }
    }
}
```

A3.7 DM-Index Model_Primary Deployment

```
namespace multilevelindex
{
    class primaryindex_class : generalindex_class
    {
        SortedDictionary<int, keyclass_class> primaryindex_ = new
SortedDictionary<int, keyclass_class>();

        public primaryindex_class()
        {
            indexID_ = configuration_class.primaryindexID;
            indexname_ = configuration_class.primaryindex;
        }
        public override bool add(service_class service, int selectedkey)
        {
            int servicekey = selectedkey;
            keyclass_class keyclass;
            if (primaryindex_.TryGetValue(servicekey, out keyclass))
            {
                if (keyclass.addtoserviceset(service))
                {
                    servicecount_++;
                    return (true);
                }
            }
            else
            {
                keyclass_class newkeyclass = new keyclass_class();
                if (newkeyclass.addtoserviceset(service))
                {
                    primaryindex_.Add(servicekey, newkeyclass);
                    servicecount_++;
                    return (true);
                }
            }
            return (false);
        }
        int selectkey(service_class service)
        {
            int key1 = -1, key2 = -1, key3 = -1;
```

```
            double key1size = -1, key3size = double.MaxValue;
            double optimumvalue =
configuration_class.calculateoptimumvalue(servicecount_ + 1);
            foreach (int inputparameter in service.inputs)
            {
                keyclass_class keyclass;
                if (primaryindex_.TryGetValue(inputparameter, out keyclass))
                {
                    double keyclasssize = keyclass.servicecount + 1;
                    if (keyclasssize <= optimumvalue)
                    {
                        if (keyclasssize > key1size)
                        {
                            key1 = inputparameter;
                            key1size = keyclasssize;
                        }
                    }
                    else                    {
                        if (key1 == -1 && key2 == -1 && keyclasssize < key3size)
                        {
                            key3 = inputparameter;
                            key3size = keyclasssize;
                        }
                    }
                }
                else
                {
                    if (key2 == -1)
                        key2 = inputparameter;
                }
            }
            if (key1 != -1)
            {
                service.key = key1;
                return (key1);
            }
            if (key2 != -1)
            {
                service.key = key2;
                return (key2);
            }
            service.key = key3;
```

```
            return (key3);
        }

        public override LinkedList<service_class> retrieval(int
parameterset)
        {
            LinkedList<service_class> retrievedsericelist = new
LinkedList<service_class>();

                keyclass_class keyclass;
                if (primaryindex_.TryGetValue(parameterset, out keyclass))
                {
                    keyclass.retrievalfromserviceset(parameterset,
retrievedsericelist);
                    Common.SetRetrievalNum(keyclass.servicecount);
                }

            return (retrievedsericelist);
        }
        public override bool delete(service_class service)
        {
            keyclass_class keyclass;
            if (primaryindex_.TryGetValue(service.key, out keyclass))
            {
                if (keyclass.deletefromserviceset(service))
                {
                    if (keyclass.servicecount == 0)
                        primaryindex_.Remove(service.key);
                    servicecount--;
                    return true;
                }
            }
            return (false);
        }
    }

}
```

A3.8 DM-IndexModel_Partial Deployment

```
namespace multilevelindex
{
    class mediumindex_class : generalindex_class
    {
        SortedDictionary<int, keyclass_class> mediumindex_ = new
SortedDictionary<int, keyclass_class>();

        public mediumindex_class()
        {
            indexID_ = configuration_class.mediumindexID;
            indexname_ = configuration_class.mediumindex;
        }

        public override bool add(service_class service, int selectedkey)
        {
            semisimilarclass_class semisimilarclass = null;
            LinkedList<keyclass_class> keyclasslist = null;
            int unkey;
            if (findequalsemisimilarclass(service, out semisimilarclass, out
keyclasslist, out unkey, mediumindex_))
            {
                if (semisimilarclass.addtoserviceset(service))
                {
                    servicecount_++;
                    return (true);
                }
            }
            keyclass_class keyclass;
            int key = selectkey(service, keyclasslist, unkey, out keyclass,
semisimilarclasscount_);
            if (key == unkey)
            {
                mediumindex_.Add(key, keyclass);
            }
            if (keyclass.addservicetonewsemisimilarclass(service,
configuration_class.mediumindexID))
            {
                servicecount_++;
                semisimilarclasscount_++;
```

```
                return (true);
            }
            return (false);
        }

        public override LinkedList<service_class> retrieval(int
parameterset)
        {
            LinkedList<service_class> servicelist = new
LinkedList<service_class>();
            keyclass_class keyclass = null;

            if (mediumindex_.TryGetValue(parameterset, out keyclass))
            {
                keyclass.retrievalfromsemisimilarclass(parameterset,
servicelist, configuration_class.mediumindexID);
                Common.SetRetrievalNum(keyclass.semisimilarclasscount);
            }

            return (servicelist);
        }

        public override bool delete(service_class service)
        {
            keyclass_class keyclass = null;
            if (mediumindex_.TryGetValue(service.key, out keyclass))
            {
                if (keyclass.deletefromsemisimilarclass(service, ref
semisimilarclasscount_, configuration_class.mediumindexID))
                {
                    if (keyclass.semisimilarclasscount == 0)
                    {
                        mediumindex_.Remove(service.key);
                    }
                    servicecount_--;
                    return (true);
                }
            }
            return (false);
        }
    }
}
```

A3.9 DNEB-Chord

```
namespace multilevelindex
{
    class Chord
    {
        private int _size;
        private int _serviceIndexID;
        private List<Node> _nodesList = new List<Node>();
        private List<int> _ipHashList = new List<int>();
        private bool _debug = true;
        private string next = "Y";
        private int num_lines = Common.GetLine();
        private int i = 1; //for screen output.
        private static int nodeCount = 0;
        public Chord(int size, int indexID)
        {
            _size = size;
            if (_size == 0)
            {
                Console.WriteLine("incorrect size,please input the correct
size");
            }
            else
            {
                _serviceIndexID = indexID;
                Common.SetSize(_size);
                initNodes(_serviceIndexID);
                setPreAndSuccNodes();
                createFigerTable();
                printNodesAndFingerTable();
            }
        }
private void initNodes(int indexID)
        {
            int idSpaceSize = Common.getIDSpace();
            int i, j;
            int k = 1;
            _debug = true;
            for (i=1;i<=8;i++)
```

```
                for(j=2;j<=26;j++)
                {
                    Address z = new Address();
                    string temp = 1.ToString() + "." + 1.ToString() + "." +
i.ToString() + "." + j.ToString();
                    z.SetIpAddress(temp);
                    z.SetMap(FHash.MapNode(z.GetIpAddress(), idSpaceSize));
                    if (_ipHashList.Contains(z.GetMap()))
                    {
                        Console.WriteLine("********Warning!!! Duplicated
ipHash was found**********************");
                        Console.ReadLine();
                    }
                    else
                    {
                        _ipHashList.Add(z.GetMap());
                    }
                    Node newNode = new Node(z, indexID);
                    newNode._ipHash = z.GetMap();
                    _nodesList.Add(newNode);
                    if (_debug)
                    {
                        Console.WriteLine("addnode complete!!" + "[" + (k++)
+ "]" + temp + " with hash: " + z.GetMap());
                        next = Console.ReadLine();
                    }
                    if (next == "n" || next == "N")
                    {
                        _debug = false;
                    }
                }
            Console.WriteLine("Node HASH (ID) before Sorted:");
            foreach (int element in _ipHashList)
            {
                Console.Write(element + ", ");
            }
            Console.ReadLine();
        }
    private void initNodes(int indexID)
    {
        int idSpaceSize = Common.getIDSpace();
        _debug = true;
```

```
        int t = 0;
        Random ro = new Random(configuration_class.getNodeRandomSeed());
        List<string> _ipList = new List<string>();
        while (t<Common.getNodeNumber())
        {
            Address z = new Address();
            string temp = ro.Next(1,255).ToString() + "." +
ro.Next(1,255).ToString() + "." + ro.Next(1, 255).ToString() + "." + ro.Next(1,
255).ToString();
            if (!_ipList.Contains(temp))
            {

                z.SetIpAddress(temp);
                z.SetMap(FHash.MapNode(z.GetIpAddress(), idSpaceSize));
                if (!_ipHashList.Contains(z.GetMap()))
                {
                    _ipList.Add(temp);
                    _ipHashList.Add(z.GetMap());
                    Node newNode = new Node(z, indexID);
                    newNode._ipHash = z.GetMap();
                    _nodesList.Add(newNode);
                    t++;
                    if (_debug)
                    {
                        Console.WriteLine("addnode complete!!" + "[" + t + "]"
+ temp + " with hash: " + z.GetMap());
                        next = Console.ReadLine();
                    }
                    if (next == "n" || next == "N")
                    {
                        _debug = false;
                    }
                 }
            }
        }
        Console.WriteLine("node hash key before sort:");
        foreach (int element in _ipHashList)
        {
            Console.Write(element + ", ");
        }
        Console.ReadLine();
    }
```

```
         private void setPreAndSuccNodes()
        {
            _nodesList.Sort();
            for (int i=1; i<_nodesList.Count-1;i++)
            {
                _nodesList[i].SetPredecessor(_nodesList[i - 1]);
                _nodesList[i].SetSuccessor(_nodesList[i + 1]);
            }
            _nodesList.First().SetPredecessor(_nodesList.Last());
            _nodesList.First().SetSuccessor(_nodesList[1]);
            _nodesList.Last().SetPredecessor(_nodesList[_nodesList.Count -
2]);
            _nodesList.Last().SetSuccessor(_nodesList.First());
        }
        private void createFigerTable()
        {
            _debug = true;
            int m = Common.getIDSpace();
            int MAX = 1 << m; // 2^m
            int length = _nodesList.Count;
            Console.WriteLine("Node HASH (ID) after Sorted in Sequence (with
IP Address):");
            foreach (Node element in _nodesList)
            {
                Console.WriteLine(element.GetAddress().GetIpAddress() + ": "
+ element._ipHash + ", ");
            }
            Console.ReadLine();
            Console.WriteLine("Node HASH (ID) after Sorted (without IP
Address):");
            foreach (Node element in _nodesList)
            {
                Console.Write(element._ipHash + ", ");
            }
            Console.ReadLine();
            foreach (Node element in _nodesList)
            {
                int i, j;
                int diff, lastLines;
                for (i = 0; i < num_lines; i++)
                {
                    element._fTable[i] = new Finger();
```

```
                    element._fTable[i].SetStartHash((element._ipHash +
(int)Math.Pow(2, i)) % MAX);
                }
                if (Common.GetCounterCircle())
                {
                    for (i = 0; i < num_lines; i++)
                    {
                        element._rTable[i] = new Finger();
                        element._rTable[i].SetStartHash((element._ipHash +
MAX - (int)Math.Pow(2, i)) % MAX);
                    }
                }

                i = 0;
                j = 1;
                lastLines = 0;
                while (i < num_lines)
                {
                    int index = _nodesList.IndexOf(element) + j;
                    if (index >= length)
                    {
                        index = index - length;
                        diff = _nodesList[index]._ipHash + MAX -
element._ipHash;
                    }
                    else
                    {
                        diff = _nodesList[index]._ipHash - element._ipHash;
                    }
                    int lines = Convert.ToInt32(Math.Floor(Math.Log(diff, 2)))
+ 1;
                    while (lastLines < lines && lastLines < num_lines)
                    {

element._fTable[lastLines].SetNode(_nodesList[index]);
                        lastLines++;
                    }
                    i = lines;
                    j++;
                    lastLines = lines;
                }
                if (Common.GetCounterCircle())
```

```
            {
                i = 0;
                j = 1;
                lastLines = 0;
                while (i < num_lines)
                {
                    int index = _nodesList.IndexOf(element) - j;
                    if (index < 0)
                    {
                        index = index + length;
                        diff = -_nodesList[index]._ipHash +
element._ipHash + MAX;
                    }
                    else
                    {
                        diff = -_nodesList[index]._ipHash +
element._ipHash;
                    }
                    int lines =
Convert.ToInt32(Math.Ceiling(Math.Log(diff, 2)));
                    while (lastLines < lines && lastLines < num_lines)
                    {
                        if (index + 1 >= length)
                        {
                            Console.WriteLine("*****index: "+ index
+"******index  over length*****");
                            index = -1;
                        }

element._rTable[lastLines].SetNode(_nodesList[index + 1]);
                        lastLines++;
                    }
                    i = lines;
                    j++;
                    lastLines = lines;
                }
            }
        }
        public void printNodesAndFingerTable()
        {
            _debug = true;
```

```
            foreach (Node element in _nodesList)
            {
                if (_debug)
                {

Console.WriteLine("**********************************************************
*****");
                    Console.WriteLine("*Node Ip: " +
element.GetAddress().GetIpAddress() + " *Node HASH (ID): " +
element.GetAddress().GetMap() + " *");
                    Console.WriteLine("    Finger Tables ");
                    Console.WriteLine("Clockwise Figure Table ");
                    Console.WriteLine("Figure" + "     |Start " + "|" +
"Successor");
                    Console.WriteLine("————————————");
                    for (int i = 0; i < num_lines; i++)
                    {
                        Console.WriteLine("Figure["+i+ "]" + "
|"+element._fTable[i].GetStartHash() + "   |" +
element._fTable[i].GetNode()._ipHash);
                    }
                    Console.WriteLine("");
                    if (Common.GetCounterCircle())
                    {
                        Console.WriteLine("Counter Clockwise Figure Table ");
                        Console.WriteLine("Figure" + "     |Start " + "|"+
"Predecessor");
                        Console.WriteLine("————————————");
                        for (int i = 0; i < num_lines; i++)
                        {
                            Console.WriteLine("Figure[" + i + "]" + "
|"+element._rTable[i].GetStartHash() + "   |" +
element._rTable[i].GetNode()._ipHash);
                        }
                    }
                    next = Console.ReadLine();
                }
                if (next == "n" || next == "N")
                {
                    _debug = false;
                }
            }
```

```
        }
        public void addKeys(service_class service)
        {
            string serviceKey;
            int serviceKeyHash;
            int nodeHash;
            serviceKey = selectKey(service).ToString();
            serviceKeyHash = FHash.MapString(serviceKey,
Common.getIDSpace());
            if (i == 1)
            {
                _debug = true;
                i = i + 1;
            }
            foreach (Node element in _nodesList)
            {
                nodeHash = element._ipHash;
                if (serviceKeyHash <= nodeHash)
                {
                    bool result = element.storeKey(serviceKeyHash, service);
                    if (!result)
                    {
                        Console.WriteLine("WARNING!!! service key: " +
serviceKeyHash + " ADDING FAILURE");
                        break;
                    }
                    if (_debug)
                    {
                        Console.WriteLine("service key: " + serviceKeyHash +
" was added into node (hash): " + nodeHash);
                        next = Console.ReadLine();
                    }
                    if (next == "n" || next == "N")
                    {
                        _debug = false;
                    }

                    break;
                }
                else if (serviceKeyHash > _nodesList.Last()._ipHash)
                {
                    bool result = _nodesList.First().storeKey(serviceKeyHash,
```

```
service);
                    if (!result)
                    {
                        Console.WriteLine("WARNING!!! service key: " +
serviceKeyHash + " ADDING FAILURE");
                        break;
                    }
                    if (_debug)
                    {
                        Console.WriteLine("service key: " + serviceKeyHash +
" was added into the first node (hash): " + _nodesList.First()._ipHash);
                        next = Console.ReadLine();
                    }
                    if (next == "n" || next == "N")
                    {
                        _debug = false;
                    }
                    break;
                }
            }
        }
        public int selectKey(service_class service)
        {
            return service.inputs.FirstOrDefault<int>();
        }
        public void showKeysInEachNode()
        {
            int serviceNum = 0;
            foreach (Node element in _nodesList)
            {
               element._multilevelIndex.getIndexDict().Keys)
                foreach (service_class i in
element._sequenceIndex.getServiceSet().getServiceSet().Values)
                {
                    Console.WriteLine("Node: " + element._ipHash + " has the
service keys: " + i.inputs.FirstOrDefault() + " with hash: " +
FHash.MapString(i.inputs.FirstOrDefault().ToString(),
Common.getIDSpace()));
                }
                next = Console.ReadLine();
                serviceNum = serviceNum +
element._sequenceIndex.getServiceSet().getServiceSet().Values.Count;
```

```
            if (next == "n" || next == "N")
            {
                break;
            }
        }
        Console.WriteLine("total service number is: " + serviceNum);
    }
    public Node findNode(int serviceKeyHash, Node node)
    {
        Node _node = node;
        Node tempNode = node;
        nodeCount++;
        int temp = 0;
        if (serviceKeyHash == _node._ipHash)
            return _node;
        else if (serviceKeyHash < _nodesList.First()._ipHash)
            return _nodesList.First();
        else if (serviceKeyHash < _node._ipHash)
            return findNode(serviceKeyHash, _node.GetPredecessor());
        else
        {
            foreach (Finger i in _node._fTable)
            {
                if (serviceKeyHash == i.GetNode()._ipHash)
                    return i.GetNode();
                else if (serviceKeyHash > i.GetNode()._ipHash)
                {
                    if (i.GetNode()._ipHash > temp)
                    {
                        temp = i.GetNode()._ipHash;
                        tempNode = i.GetNode();
                    }
                }
            }
            if (serviceKeyHash <= tempNode.GetSuccessor()._ipHash)
                return tempNode.GetSuccessor();
            else if (serviceKeyHash > tempNode.GetSuccessor()._ipHash &&
tempNode.GetSuccessor()._ipHash > tempNode._ipHash)
                return findNode(serviceKeyHash, tempNode);
            else if (serviceKeyHash > tempNode.GetSuccessor()._ipHash &&
tempNode.GetSuccessor()._ipHash < tempNode._ipHash)
                return tempNode.GetSuccessor();
```

```
            }
            return null;
        }
        public Node findNode_counterClockwise(int serviceKeyHash, Node node)
        {
            Node _node = node;
            Node tempNode = node;
            nodeCount++;
            int temp = 1 << Common.getIDSpace();
            foreach (Finger i in _node._rTable)
            {
                if (serviceKeyHash == i.GetNode()._ipHash)
                    return i.GetNode();
                else if (serviceKeyHash < i.GetNode()._ipHash)
                {
                   if (i.GetNode()._ipHash < temp)
                    {
                        temp = i.GetNode()._ipHash;
                        tempNode = i.GetNode();
                    }
                }
            }
            if (serviceKeyHash > tempNode.GetPredecessor()._ipHash)
                return tempNode;
            else if (serviceKeyHash == tempNode.GetPredecessor()._ipHash)
                return tempNode.GetPredecessor();
            else if (serviceKeyHash < tempNode.GetPredecessor()._ipHash &&
tempNode.GetPredecessor()._ipHash < tempNode._ipHash)
                return findNode_counterClockwise(serviceKeyHash,
tempNode.GetPredecessor());
            else if (tempNode.GetPredecessor()._ipHash > tempNode._ipHash)
                return tempNode;
            else
                return findNode_counterClockwise(serviceKeyHash, tempNode);
        }
        public bool retriveService(SortedSet<int> retrievalrequest)
        {
            string serviceKey;
            int serviceKeyHash;
            _debug = true;
            bool ifFound = false;
            Node initNode = _nodesList[0];
```

```
            Node node;
            bool result;
            int midNodeHash =
_nodesList.ElementAt(Convert.ToInt32(Math.Ceiling(_nodesList.Count
/2.0)))._ipHash;
           foreach (int request in retrievalrequest)
            {
                serviceKey = request.ToString();
                serviceKeyHash = FHash.MapString(serviceKey,
Common.getIDSpace());
                ifFound = false;
                if (Common.GetCounterCircle())
                {
                    if (serviceKeyHash < midNodeHash)
                    {
                        initNode = _nodesList.First();
                        nodeCount = 0;
                        node = findNode(serviceKeyHash, initNode);
                    }
                    else
                    {
                        initNode = _nodesList.Last();
                        nodeCount = 0;
                        node = findNode_counterClockwise(serviceKeyHash,
initNode);
                    }
                    result = node.retriveKey(request);
                }
                else
                {
                    nodeCount = 0;
                    node = findNode(serviceKeyHash, initNode);
                    result = node.retriveKey(request);
                }
                if (result)
                {
                    ifFound = true;
                    Common.setNodeCounts(nodeCount);
                }
               if (_debug && ifFound)
                {
                    Console.WriteLine("Service with Key(HASH): " + request +
```

```
"("+serviceKeyHash+")" + " was found at node position: " +
_nodesList.IndexOf(node)+1 +" with node hash: " + node._ipHash);
                    Console.WriteLine("This key was searched over " + nodeCount
+ " nodes");
                    next = Console.ReadLine();
                }
                if (next == "n" || next == "N")
                {
                    _debug = false;
                }
                if (ifFound)
                break;
            }
            if (!ifFound)
            {
                Console.WriteLine("WARNING!!! service was not found for this
request!!!");
                return false;
            }
            else
            {
                Console.WriteLine("Found the service key!");
                return true;
            }

        }
    }
}
```

A3.10 Test Performances

```
namespace multilevelindex
{
    using testseting_class = results_class;

    class testsetingsinglenode_class
    {
        protected int parametersetsize_ =
configuration_class.defaultparametersetsize;
        public int parametersetsize
        {
            get { return (parametersetsize_); }
            set { parametersetsize_ = value; }
        }
        protected int servicesetsize_ =
configuration_class.defaultservicesetsize;
        public int servicesetsize
        {
            get { return (servicesetsize_); }
            set { servicesetsize_ = value; }
        }
        protected int parametercountperservice_ =
configuration_class.defaultparametercountperservice;
        public int parametercountperservice
        {
            get { return (parametercountperservice_); }
            set { parametercountperservice_ = value; }
        }
        protected int retrievalcount_ =
configuration_class.defaultretrievalcount;
        public int retrievalcount
        {
            get { return (retrievalcount_); }
            set { retrievalcount_ = value; }
        }
        protected int retrievalparametersetsize_ =
configuration_class.defaultretrievalparametersetsize;
        public int retrievalparametersetsize
        {
            get { return (retrievalparametersetsize_); }
```

```
            set { retrievalparametersetsize_ = value; }
        }
        protected int numberofnode_ =
configuration_class.defaultnumberofnode;
        public int numberofnode
        {
            get { return (numberofnode_); }
            set { numberofnode_ = value; }
        }
        protected int defaulnumberoffingertable_ =
configuration_class.defaultnumberoffingertable;
        public int numberoffingertable
        {
            get { return (defaulnumberoffingertable_); }
            set { defaulnumberoffingertable_ = value; }
        }
        protected int defaulnumberofhash_ =
configuration_class.defaultnumberofhash;
        public int defaulnumberofhash
        {
            get { return (defaulnumberofhash_); }
            set { defaulnumberofhash_ = value; }
        }
        protected int testedindexID_ = -1;
        protected generalindex_class testedindex_;
        public generalindex_class testedindex
        {
            get { return (testedindex_); }
        }
        public testsetingsinglenode_class() { }
        public testsetingsinglenode_class(int indexid)
        {
            testedindex_ = setnewgeneralindex(indexid);
        }
        public testsetingsinglenode_class(testsetingsinglenode_class t1)
        {
            parametersetsize_ = t1.parametersetsize;
            servicesetsize_ = t1.servicesetsize;
            parametercountperservice_ = t1.parametercountperservice;
            retrievalcount_ = t1.retrievalcount;
            retrievalparametersetsize_ = t1.retrievalparametersetsize;
            testedindexID_ = t1.testedindexID_;
```

```
            testedindex_ = setnewgeneralindex(testedindexID_);
        }
        generalindex_class setnewgeneralindex(int indexID)
        {
            testedindexID_ = indexID;
            switch (testedindexID_)
            {
                case configuration_class.primaryindexID:
                    testedindex_ = new primaryindex_class();
                    break;
                case configuration_class.mediumindexID:
                    testedindex_ = new mediumindex_class();
                    break;
                case configuration_class.multilevelindexID:
                    testedindex_ = new multilevelindex_class();
                    break;
                case configuration_class.invertedindexID:
                    testedindex_ = new invertedindex_class();
                    break;
                case configuration_class.sequenceindexID:
                    testedindex_ = new sequencedindex_class();
                    break;
            }
          return (testedindex_);
        }
          public result_class test()
        {
            Stopwatch st = new Stopwatch();
            LinkedList<service_class> servicelistset = new
LinkedList<service_class>();
            LinkedList<SortedSet<int>> retrievalrequestlistset = new
LinkedList<SortedSet<int>>();
            Console.WriteLine(testedindex_.indexname + "，表" +
testedindexID_);
            Console.WriteLine("S：" + servicesetsize_ + "， P：" +
parametersetsize_ + "， NPS：" + parametercountperservice_
                + "， NPR：" + retrievalparametersetsize_ +"， NN: "+
numberofnode_+"， NFT: "
                + defaulnumberoffingertable_+"， NHB: " +
defaulnumberofhash_+"， NR：" + retrievalcount_);
            Console.WriteLine("Setting up Service Set，Size：" +
servicesetsize_);
```

```
            DateTime starttime = DateTime.Now;
            for (int i = 0; i < servicesetsize; i++)
            {
                service_class service = new service_class();
                service.fillinputsandoutputs(parametercountperservice,
parametercountperservice, parametersetsize, parametersetsize);
                servicelistset.AddLast(service);
                }
            DateTime endtime = DateTime.Now;
            Console.WriteLine("Time: " + (endtime -
starttime).TotalMilliseconds + " ms");
            Console.WriteLine("Setting up Node HASH (ID) with IP Address");

            starttime = DateTime.Now;
            Common.SetCounterCircle(true);
            Chord chord = new Chord(Common.getNodeNumber(), testedindexID_);
            endtime = DateTime.Now;
            Console.WriteLine("Time: " + (endtime -
starttime).TotalMilliseconds + " ms");
            Console.WriteLine("Adding Service to Node with Selected Input
Parameter (Key) HASH");
            result_class result = new result_class();
            starttime = DateTime.Now;
            foreach(service_class service in servicelistset)
            {
                chord.addKeys(service);
            }
            endtime = DateTime.Now;
            double time = (endtime - starttime).TotalMilliseconds;
            result.additiontime =time;
            Console.WriteLine("Time: " + time+" ms");
            Console.WriteLine("Setting up Retrieval, Size: " +
retrievalcount_);
            starttime = DateTime.Now;
            for (int j = 0; j < retrievalcount; j++)
            {
                retrievalrequest_class retrievalrequest = new
retrievalrequest_class();

retrievalrequestlistset.AddLast(retrievalrequest.request(retrievalparamet
ersetsize, parametersetsize));
            }
```

```
            endtime = DateTime.Now;
            Console.WriteLine("Time： " + (endtime -
starttime).TotalMilliseconds + " ms");
            if (Common.GetCounterCircle())
            {
                Common.setNodeCounts(0);
                Console.WriteLine("");
                Console.WriteLine("");
                Console.WriteLine("Retrieving with BiChord)");
                Common.setRetriveNum(0);
                starttime = DateTime.Now;
                result.accessedservicecount = 0;
                result.starttimeBiChord = starttime.Ticks;
                foreach (SortedSet<int> retrievalrequest in
retrievalrequestlistset)
                {
                    bool ifFound = chord.retriveService(retrievalrequest);
                    result.nodeCounts_BiChord += Common.getNodeCounts();
                    result.accessedservicecount += Common.GetRetrievalNum();
                    if (ifFound)
                    Common.setRetriveNum(Common.getRetriveNum() + 1);
                    }
                endtime = DateTime.Now;
                result.endtimeBiChord = endtime.Ticks;
                long Ticks = endtime.Ticks - starttime.Ticks;
                result.retrievaltimeBiChord = Ticks/10;
            Common.setNodeCounts(0);
            Console.WriteLine("");
            Console.WriteLine("");
            Console.WriteLine("***************");
            Console.WriteLine("Retrieving with Chord");
            Common.setRetriveNum(0);
            result.accessedservicecount = 0;
            starttime = DateTime.Now;
            result.starttimeChord = starttime.Ticks;
            foreach (SortedSet<int> retrievalrequest in
retrievalrequestlistset)
            {
                bool ifFound = chord.retriveService(retrievalrequest);
                result.nodeCounts += Common.getNodeCounts();
                result.accessedservicecount += Common.GetRetrievalNum();
                if (ifFound)
```

```
                    Common.setRetriveNum(Common.getRetriveNum()+1);
            }
            endtime = DateTime.Now;
            result.endtimeChord = endtime.Ticks;
            long elapsedTicks = endtime.Ticks - starttime.Ticks;
             TimeSpan elapsedSpan = new TimeSpan(elapsedTicks);
            result.retrievaltime = elapsedTicks/10;
            result.retrievedservicecount = Common.getRetriveNum();
            Console.WriteLine("Time: " + elapsedTicks/10 + " us");
            Console.WriteLine("Found: " + Common.getRetriveNum() + "
services");
            return result;
            }
    }
    class result_class
    {
        protected double additiontime_;
        protected long retrievaltime_;
        protected long retrievaltimeBiChord_;
        protected string retrievaltimeBiChords_;
        protected double semisimilarclasscount_;
        protected double accesssedservicecount_;
        protected int retrievedservicecount_;
        protected int retrievedservicecountBiChord_;
        protected int nodeCounts_;
        protected int nodeCountsBiChord_;
        protected long starttimeBiChord_;
        protected long endtimeBiChord_;
        protected long starttimeChord_;
        protected long endtimeChord_;
      public int nodeCounts_BiChord
        {
            get { return nodeCountsBiChord_; }
            set { nodeCountsBiChord_ = value; }
        }
        public int nodeCounts
        {
            get { return nodeCounts_; }
            set { nodeCounts_ = value; }
        }
       public double additiontime
        {
```

```
    get { return additiontime_; }
    set { additiontime_ = value; }
 }
 public long retrievaltime
 {
    get { return retrievaltime_; }
    set { retrievaltime_ = value; }
 }
 public long retrievaltimeBiChord
 {
    get { return retrievaltimeBiChord_; }
    set { retrievaltimeBiChord_ = value; }
 }
 public long starttimeBiChord
 {
     get { return starttimeBiChord_; }
     set { starttimeBiChord_ = value; }
  }
public long endtimeBiChord
 {
     get { return endtimeBiChord_; }
     set { endtimeBiChord_ = value; }
  }
public long starttimeChord
 {
     get { return starttimeChord_; }
     set { starttimeChord_ = value; }
  }
 public long endtimeChord
 {
     get { return endtimeChord_; }
     set { endtimeChord_ = value; }
  }
 public string retrievaltimeBiChords
 {
    get { return retrievaltimeBiChords_; }
    set { retrievaltimeBiChords_ = value; }
 }
 public double semisimilarclasscount
 {
    get { return semisimilarclasscount_; }
    set { semisimilarclasscount_ = value; }
```

```
        }
        public double accessedservicecount
        {
            get { return accesssedservicecount_; }
            set { accesssedservicecount_ = value; }
        }
        public int retrievedservicecountBiChord
        {
            get { return retrievedservicecountBiChord_; }
            set { retrievedservicecountBiChord_ = value; }
        }
        public int retrievedservicecount
        {
            get { return retrievedservicecount_; }
            set { retrievedservicecount_ = value; }
        }
     }
    class resultnode_class : result_class
    {
        int variable_;
        public int Variable
        {
            get { return variable_; }
            set { variable_ = value; }
        }
        public resultnode_class() { }
        public resultnode_class(int variablep, result_class result)
        {
            variable_ = variablep;
            additiontime_ = result.additiontime;
            retrievaltime_ = result.retrievaltime;
            semisimilarclasscount_ = result.semisimilarclasscount;
            accesssedservicecount_ = result.accessedservicecount;
            retrievedservicecount_ = result.retrievedservicecount;
            nodeCounts_ = result.nodeCounts;
            nodeCountsBiChord_ = result.nodeCounts_BiChord;
            retrievedservicecountBiChord_ =
result.retrievedservicecountBiChord;
            retrievaltimeBiChord_ = result.retrievaltimeBiChord;
            starttimeBiChord_= result.starttimeBiChord;
            endtimeBiChord_ = result.endtimeBiChord;
            starttimeChord_ = result.starttimeChord;
```

```
            endtimeChord_ = result.endtimeChord;
          }
    }
    class results_class : testsetingsinglenode_class
    {
        int testedItem_=-1;
        public int testedItem
        {
            get { return (testedItem_); }
            set { testedItem_ = value; }
        }
        int[] variablearray_;
        public int[] variablearray
        {
            get { return (variablearray_); }
            set { variablearray_ = value; }
        }
        LinkedList<resultnode_class> resultelist_;
        public results_class(int indexID)
        {
            testedindexID_ = indexID;
        }
        public LinkedList<resultnode_class> testseries()
        {
            resultelist_ = new LinkedList<resultnode_class>();
            foreach (int count in variablearray_)
            {
                testsetingsinglenode_class testsetingsinglenodenow = new
testsetingsinglenode_class(this);
                switch (testedItem_)
                {
                    case configuration_class.testservicesetsize:
                        testsetingsinglenodenow.servicesetsize = count;
                        break;
                    case configuration_class.testparametercountperservice:
                        testsetingsinglenodenow.parametercountperservice =
count;
                        break;
                    case configuration_class.testretrievalparametersetsize:
                        testsetingsinglenodenow.retrievalparametersetsize =
count;
                        break;
```

```
                    case configuration_class.testparametersetsize:
                        testsetingsinglenodenow.parametersetsize = count;
                        break;

                    case configuration_class.testnumberofnode:
                        testsetingsinglenodenow.numberofnode = count;
                        break;
                    case configuration_class.testretrievalcount:
                        testsetingsinglenodenow.retrievalcount = count;
                        break;
                }
                resultelist_.AddLast(new resultnode_class(count,
testsetingsinglenodenow.test()));
            }
            return (resultelist_);
        }
        public bool writetoexcel(string filepath, int sheetnumber)
        {
            Microsoft.Office.Interop.Excel.Application excel = null;
            try
            {
                excel = new Microsoft.Office.Interop.Excel.Application();
                Microsoft.Office.Interop.Excel.Workbook xlbook =
excel.Workbooks.Open(filepath, Missing.Value,
                    Missing.Value, Missing.Value, Missing.Value,
Missing.Value, Missing.Value, Missing.Value, Missing.Value, Missing.Value,
                    Missing.Value, Missing.Value, Missing.Value);
                if (xlbook.ReadOnly)
                {
                    Console.WriteLine("readonly, File is being used");
                    return (false);
                }
                 Microsoft.Office.Interop.Excel.Worksheet xlsheet =
xlbook.Worksheets[sheetnumber];
                xlsheet.Cells[1, 1] = "Index Model";
                xlsheet.Cells[2, 1] =
generalindex_class.getindexname(testedindexID_);
                xlsheet.Cells[1, 2] = "S";
                xlsheet.Cells[2, 2] = servicesetsize_;
                xlsheet.Cells[1, 3] = "P";
                xlsheet.Cells[2, 3] = parametersetsize_;
                xlsheet.Cells[1, 4] = "NPS";
```

```
                xlsheet.Cells[2, 4] = parametercountperservice_;
                xlsheet.Cells[1, 5] = "NPR";
                xlsheet.Cells[2, 5] = retrievalparametersetsize_;
                xlsheet.Cells[1, 6] = "NR";
                xlsheet.Cells[2, 6] = retrievalcount_;
                xlsheet.Cells[1, 7] = "NN";
                xlsheet.Cells[2, 7] = numberofnode_;
                xlsheet.Cells[1, 8] = "NFT";
                xlsheet.Cells[2, 8] = defaulnumberoffingertable_;
                xlsheet.Cells[1, 9] = "NHB";
                xlsheet.Cells[2, 9] = defaulnumberofhash_;
                xlsheet.Cells[3, 1] = "S";
                xlsheet.Cells[3, 15] = "P";
                xlsheet.Cells[3, 29] = "NPS";
                xlsheet.Cells[3, 43] = "NPR";
                xlsheet.Cells[3, 57] = "NN";
                xlsheet.Cells[3, 71] = "NR";
              for (int i = 0; i < 6; i++)
                {
                    xlsheet.Cells[3, 2 + 14 * i] = "Addtion Time(ms)";
                    xlsheet.Cells[3, 3 + 14 * i] = "C-Retrieval Time(us)";
                    xlsheet.Cells[3, 4 + 14 * i] = "C-Start Time (Ticks)";
                    xlsheet.Cells[3, 5 + 14 * i] = "C-End Time (Ticks)";
                    xlsheet.Cells[3, 6 + 14 * i] = "BC-Retrieval Time (us)";
                    xlsheet.Cells[3, 7 + 14 * i] = "BC-Start Time(Ticks)";
                    xlsheet.Cells[3, 8 + 14 * i] = "BC-End Time(Ticks)";
                    xlsheet.Cells[3, 9 + 14 * i] = "Retrieved Service Count";
                    xlsheet.Cells[3, 10 + 14 * i] = "BC-Retrieved Service
Count";
                    xlsheet.Cells[3, 11 + 14 * i] = "Hop Count";
                    xlsheet.Cells[3, 12 + 14 * i] = "BC-Hop Count";
                    xlsheet.Cells[3, 13 + 14 * i] = "AccesssedS Service Count";
                 }
               int col = -1, row = 4;
                if (testedItem_ == configuration_class.testservicesetsize)
                    col = 1 + 14 * 0;
                if (testedItem_ == configuration_class.testparametersetsize)
                    col = 1 + 14 * 1;
                if (testedItem_ ==
configuration_class.testparametercountperservice)
                    col = 1 + 14 * 2;
                if (testedItem_ ==
```

```
configuration_class.testretrievalparametersetsize)
                    col = 1 + 14 * 3;
                if (testedItem_ == configuration_class.testnumberofnode)
                    col = 1 + 14 * 4;
if (testedItem_ == configuration_class.testretrievalcount)
                    col = 1 + 14 * 5;
foreach (resultnode_class resultnod in resultelist_)
                {
                    xlsheet.Cells[row, col] = resultnod.Variable;
                    xlsheet.Cells[row, col + 1] = resultnod.additiontime;
                    xlsheet.Cells[row, col + 2] = resultnod.retrievaltime;
                    xlsheet.Cells[row, col + 3] = resultnod.starttimeChord;
                    xlsheet.Cells[row, col + 4] = resultnod.endtimeChord;
                    xlsheet.Cells[row, col + 5] =
resultnod.retrievaltimeBiChord;
                    xlsheet.Cells[row, col + 6] = resultnod.starttimeBiChord;
                    xlsheet.Cells[row, col + 7] = resultnod.endtimeBiChord;
                    xlsheet.Cells[row, col + 8] =
resultnod.retrievedservicecount;
                    xlsheet.Cells[row, col + 9] =
resultnod.retrievedservicecountBiChord;
                    xlsheet.Cells[row, col + 10] = resultnod.nodeCounts;//By
Miao
                    xlsheet.Cells[row, col + 11] =
resultnod.nodeCounts_BiChord;
                    xlsheet.Cells[row, col + 12] =
resultnod.accessedservicecount;
                   row++;
                }
                xlbook.Save();
                excel.Workbooks.Close();
                excel.Quit();
                return (true);
            }
            catch (System.Runtime.InteropServices.COMException excp)
            {
                Console.WriteLine(excp.Message);
            }
            finally
            {
                if (excel.Workbooks != null)
                    excel.Workbooks.Close();
```

```
            if (excel != null)
                excel.Quit();
        }
        return (false);
    }
}
class testperformance_class
{
    public static void testperformace(int testedindexID, string
resultfilepath, int testeditem,
        int initialvalue, int length, int row)
    {
        testseting_class testseting = new
testseting_class(testedindexID);
        Console.WriteLine("Test Starting: " +
generalindex_class.getindexname(testedindexID));

Console.WriteLine("TestStarting"+gettesteditemname(testeditem));
        testseting.testedItem = testeditem;
        testseting.variablearray =
configuration_class.intarray(initialvalue, length, row);
        testseting.testseries();
        testseting.writetoexcel(resultfilepath, testedindexID);
    }
    public static string gettesteditemname(int testeditem)
    {
        switch (testeditem)
        {
            case configuration_class.testservicesetsize:
                return (configuration_class.servicesetsize);
            case configuration_class.testparametercountperservice:
                return (configuration_class.parametercountperservice);
            case configuration_class.testretrievalparametersetsize:
                return (configuration_class.retrievalparametersetsize);
            case configuration_class.testparametersetsize:
                return (configuration_class.parametersetsize);
            case configuration_class.testnumberofnode:
                return (configuration_class.numberofnode);
            case configuration_class.testretrievalcount:
                return (configuration_class.retrievalcount)
      }
        return (null);
```

```
        }
    }

}
```

A3.11 Save Results to Excel File

```
namespace multilevelindex
{
    class excel_class
    {
        public void Intoexcel(string filepath, int sheetnumber, double[]
sourcedata, int row, int col)
        {
            Microsoft.Office.Interop.Excel.Application excel = new
Microsoft.Office.Interop.Excel.Application();
            Microsoft.Office.Interop.Excel.Workbook xlbook =
excel.Workbooks._Open(filepath,
Missing.Value,Missing.Value,Missing.Value,Missing.Value
            ,Missing.Value,Missing.Value,Missing.Value,Missing.Value ,Miss
ing.Value,Missing.Value,Missing.Value,Missing.Value);
            Microsoft.Office.Interop.Excel.Worksheet xlsheet =
(Microsoft.Office.Interop.Excel.Worksheet)xlbook.Worksheets[sheetnumber];

            int m = row;
            for (int t = 0; t < sourcedata.Length; t++)
            {
                excel.Cells[m, col] = sourcedata[t];
                m++;
            }
            xlbook.Save();
            excel.Workbooks.Close();
            excel.Quit();
        }
        public double[] outexcel(string filepath, int sheetnumber, int row,
int col)
        {
            Microsoft.Office.Interop.Excel.Application excel = new
Microsoft.Office.Interop.Excel.Application();
            Microsoft.Office.Interop.Excel.Workbook xlbook =
excel.Workbooks._Open("filepath", Missing.Value, Missing.Value,
Missing.Value, Missing.Value
            , Missing.Value, Missing.Value, Missing.Value, Missing.Value,
Missing.Value, Missing.Value, Missing.Value, Missing.Value);
            Microsoft.Office.Interop.Excel.Worksheet xlsheet =
(Microsoft.Office.Interop.Excel.Worksheet)xlbook.Worksheets[sheetnumber];
```

```
            double[] p = new double [1000000];
            int t = 0;
            for (int m = 1; m < 100; m++)
            {
                for (int n = 1; n < 100; n++)
                {
                    Microsoft.Office.Interop.Excel.Range rng2 =
(Microsoft.Office.Interop.Excel.Range)xlsheet.Cells[row, col];
                    p[t++] = rng2.Value2;
                 }
            }
            xlbook.Save();
            xlsheet = null;
            xlbook = null;
            excel.Quit();
            excel = null;
            GC.Collect();
            return p;
        }

        public static bool isclosedandok(string filepath)
        {
            Microsoft.Office.Interop.Excel.Application excel = null;
            try
            {
                excel = new Microsoft.Office.Interop.Excel.Application();
                Microsoft.Office.Interop.Excel.Workbook xlbook =
excel.Workbooks.Open(filepath, Missing.Value,
                    Missing.Value, Missing.Value, Missing.Value,
Missing.Value, Missing.Value, Missing.Value, Missing.Value, Missing.Value,
                    Missing.Value, Missing.Value, Missing.Value);
if (xlbook.ReadOnly)
                {
                    Console.WriteLine("readonly, File is being used! ! ");
                    return (false);
                }
            }
            catch (System.Runtime.InteropServices.COMException excp)
            {
                Console.WriteLine(excp.Message);
                return (false);
            }
```

```
            finally
            {
                if (excel.Workbooks != null)
                    excel.Workbooks.Close();
                if (excel != null)
                    excel.Quit();
            }
            return (true);
        }
    }

}
```