A SCALABLE AND FLEXIBLE UNSTRUCTURED SEARCH SYSTEM AND DISTRIBUTED DATA STRUCTURES FOR PEER-TO-PEER NETWORKS

By

TAE WOONG CHOI

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY UNIVERSITY OF FLORIDA 2010
ACKNOWLEDGMENTS

There are many people who are responsible for my research. Most of all I would like to express my gratitude to my supervisor, Dr. Oscar Boykin, for his patient guidance and invaluable advice, for numerous discussions and encouragement throughout the course of the research.

I would also like to thank all the members of my advisory committee, Dr. Renato Figueiredo, Dr. Tao Li, and Dr. William Hager, for their valuable time and interest in serving on my supervisory committee.

And I am indebted to all the members of ACIS (Advanced Computing and Information Systems) for their companionship and support throughout the time spent working on my research.

Finally, I would like to express my greatest gratitude to my family especially my wife, Yeonhwa, for her relentless support and love.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>3</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>7</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>8</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>11</td>
</tr>
</tbody>
</table>

**CHAPTER**

1 INTRODUCTION | 13

2 LITERATURE REVIEW | 17

2.1 Network Size Estimation on Structured P2P Systems | 17

2.2 Search in P2P Systems | 19

3 NETWORK SIZE ESTIMATION IN PEER-TO-PEER NETWORKS | 23

3.1 Introduction | 23

3.2 Network Size Estimation in P2P Networks: Theoretical Background | 25

3.2.1 The Algorithm | 25

3.2.2 Node Joining Algorithm for More Evenly Distributed Network | 27

3.2.3 Communication Cost | 28

3.3 Simulation Results | 29

3.3.1 Simulation Methodology | 29

3.3.2 Network Topology | 29

3.3.3 Simulation Results | 29

3.4 Conclusion | 31

4 DEETO: SCALABLE UNSTRUCTURED SEARCH BUILT ON A STRUCTURED OVERLAY | 38

4.1 Introduction | 38

4.2 The Deetoo Search Algorithm | 41

4.2.1 An Unstructured “Hash” Table | 41

4.2.2 A Distributed Unstructured Table | 42

4.2.3 Bounded Broadcast | 44

4.2.4 Data Insertion and Query | 45

4.2.5 Node Joins, Leaves, and Stabilization | 46

4.2.6 Estimating Network Size | 47

4.3 Analysis of Deetoo | 48

4.3.1 Exact Analysis for Success Probability | 49

4.3.2 Search Time | 52

4.3.3 Stabilization Cost | 54
### LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-1</td>
<td>Statistics of estimated network size (random address selection)</td>
</tr>
<tr>
<td>3-2</td>
<td>Statistics of estimated network size (even address selection)</td>
</tr>
<tr>
<td>5-1</td>
<td>Latency</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>3-1</td>
<td>Tree generation by bounded broadcasting</td>
</tr>
<tr>
<td>3-2</td>
<td>Address selection of newly joined node</td>
</tr>
<tr>
<td>3-3</td>
<td>Mean estimation with actual size of 5000 nodes</td>
</tr>
<tr>
<td>3-4</td>
<td>Mean estimation with actual size of 500000 nodes</td>
</tr>
<tr>
<td>3-5</td>
<td>Standard deviation of each estimation (N=5000)</td>
</tr>
<tr>
<td>3-6</td>
<td>Standard deviation of each estimation (N=5000000)</td>
</tr>
<tr>
<td>3-7</td>
<td>Communication cost with 5000 nodes</td>
</tr>
<tr>
<td>3-8</td>
<td>Communication cost with 500000 nodes</td>
</tr>
<tr>
<td>3-9</td>
<td>Communication cost vs. MSE with log-estimation (actual network size: 50,000)</td>
</tr>
<tr>
<td>4-1</td>
<td>The caching and querying space</td>
</tr>
<tr>
<td>4-2</td>
<td>Virtual ring 1 for caching</td>
</tr>
<tr>
<td>4-3</td>
<td>Virtual ring 2 for querying</td>
</tr>
<tr>
<td>4-4</td>
<td>The complete searching space</td>
</tr>
<tr>
<td>4-5</td>
<td>Tree generation by bounded broadcasting</td>
</tr>
<tr>
<td>4-6</td>
<td>Object injection</td>
</tr>
<tr>
<td>4-7</td>
<td>Bounded broadcast in the range</td>
</tr>
<tr>
<td>4-8</td>
<td>Completion of object replication</td>
</tr>
<tr>
<td>4-9</td>
<td>Initiation of a query for object o</td>
</tr>
<tr>
<td>4-10</td>
<td>Bounded broadcasting within Q</td>
</tr>
<tr>
<td>4-11</td>
<td>Object retrieval n(o)</td>
</tr>
<tr>
<td>4-12</td>
<td>Estimated network size distribution</td>
</tr>
<tr>
<td>4-13</td>
<td>Probability of long-range connection</td>
</tr>
<tr>
<td>4-14</td>
<td>Query success probability</td>
</tr>
<tr>
<td>4-15</td>
<td>Query miss rate</td>
</tr>
<tr>
<td>4-16</td>
<td>Communication cost for query</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>4-17 Search time</td>
<td>68</td>
</tr>
<tr>
<td>4-18 Success rate comparison</td>
<td>68</td>
</tr>
<tr>
<td>4-19 State transition diagram</td>
<td>69</td>
</tr>
<tr>
<td>4-20 Success probability under churn</td>
<td>69</td>
</tr>
<tr>
<td>4-21 Stabilization cost</td>
<td>70</td>
</tr>
<tr>
<td>4-22 Sequential search</td>
<td>70</td>
</tr>
<tr>
<td>5-1 P2P (Brunet) network overview</td>
<td>86</td>
</tr>
<tr>
<td>5-2 MapReduce function modules for query system</td>
<td>87</td>
</tr>
<tr>
<td>5-3 Simple example of map-reduce tree</td>
<td>87</td>
</tr>
<tr>
<td>5-4 Estimated network size ((\alpha = 1.0))</td>
<td>88</td>
</tr>
<tr>
<td>5-5 Estimated network size ((\alpha = 3.0))</td>
<td>88</td>
</tr>
<tr>
<td>5-6 Query success probability ((\alpha = 1.0))</td>
<td>89</td>
</tr>
<tr>
<td>5-7 Query success probability ((\alpha = 3.0))</td>
<td>89</td>
</tr>
<tr>
<td>5-8 Latency in terms of tree depth ((\alpha = 1.0))</td>
<td>90</td>
</tr>
<tr>
<td>5-9 Latency in terms of tree depth ((\alpha = 3.0))</td>
<td>90</td>
</tr>
<tr>
<td>5-10 Measured Latency</td>
<td>91</td>
</tr>
<tr>
<td>5-11 Communication cost ((\alpha = 1.0))</td>
<td>91</td>
</tr>
<tr>
<td>5-12 Communication cost ((\alpha = 3.0))</td>
<td>92</td>
</tr>
<tr>
<td>5-13 Total bytes transferred for each query</td>
<td>92</td>
</tr>
<tr>
<td>5-14 Communication cost for caching vs. query success probability ((\alpha = 1.0))</td>
<td>93</td>
</tr>
<tr>
<td>5-15 Communication cost for caching vs. query success probability ((\alpha = 3.0))</td>
<td>93</td>
</tr>
<tr>
<td>5-16 Load distribution</td>
<td>94</td>
</tr>
<tr>
<td>5-17 The number of cached objects per node</td>
<td>94</td>
</tr>
<tr>
<td>6-1 Example 2-dimensional 4-by-4 space</td>
<td>108</td>
</tr>
<tr>
<td>6-2 One node in the box</td>
<td>109</td>
</tr>
<tr>
<td>6-3 Two nodes in the box</td>
<td>109</td>
</tr>
<tr>
<td>6-4 Three nodes in the box</td>
<td>109</td>
</tr>
</tbody>
</table>
A SCALABLE AND FLEXIBLE UNSTRUCTURED SEARCH SYSTEM AND DISTRIBUTED DATA STRUCTURES FOR PEER-TO-PEER NETWORKS

By

Tae Woong Choi

August 2010

Peer-to-Peer concepts provide new possibilities. Many large-scale Internet services are already introduced with the help of P2P networks. However, The P2P approach to systems design has not made a big impact in practice because designing a new system with P2P concepts is still more of a research problem than a software development problem. The main goal of this dissertation is to provide components that can be combined to build new systems easily. The use of P2P techniques in system design is increased by solving a number of problems that currently present barriers. To address the problems in developing P2P systems, the dissertation focuses on four principal goals: (a) building efficient network size-estimation algorithms (b) modeling and evaluating an efficient general search system (c) testing the search system in a real P2P environment, and (d) designing a new search technique apt for P2P database services.

To address Goal (a), I propose four network size estimation algorithms. The accuracy of network size algorithm has a great impact on several applications on P2P networks including data retrieval. The methods utilize node density within various lengths in a network. The node density makes one estimate of the number of nodes in the whole system by $\hat{N} = LD$, where $\hat{N}$, $L$, and $D$ represent the estimated network size, the length of address space, and the node density, respectively. Each estimation method requires different communication costs and achieves different performance. Thus, a user can select one of the methods with respect to his purpose and expected performance.
For the pursuit of goal (b), I present Deetoo, an algorithm to perform completely general queries on a P2P network. My interest lies in building efficient unstructured P2P networks. Stated differently, I can apply the proposed model for any type of query without the necessity of mapping the queries onto a fixed DHT structure. Deetoo leverages one-dimensional routable small-world networks, and multicasting trees built on those networks, to build an unstructured query system which can support general queries, such as high-dimensional proximity queries or regular expression matching.

Lastly, this dissertation introduces how to apply P2P unstructured search to P2P database systems. The most important feature of database systems is that the search engine has to always return a correct answer. Deetoo is not sufficient for database systems due to the randomized nature. Thus, I propose Exact Deetoo, which is similar to Deetoo, but Exact Deetoo achieves 100% query success probability. The total communication cost for both caching and querying is not controllable in Exact Deetoo; however, the trade-off between caching cost and the querying cost can be managed by assigning more columns than rows or vice versa. The correctly maintained network decides the cost to achieve 100% success probability. There exists a trade-off between the exact response and the higher maintenance cost for a network structure in Exact Deetoo.

Each of the proposed algorithms are carefully tested and evaluated by simulation, experiments, as well as theoretical proofs. The proposed algorithms are applicable to be built on top of many existing P2P overlays and make it easy for developers to adopt each component.
CHAPTER 1
INTRODUCTION

Peer-to-Peer (P2P) systems are overlay networks in which nodes share resources, such as memory, CPU power, and bandwidth. Each node in a P2P system operates simultaneously as both a server and a client in a distributed fashion. Nodes navigate the underlying network without knowledge of the global structure. Since the success of applications like Napster and Gnutella, P2P file sharing systems have become some of the most common Internet applications. An Internet study reported that over 73% of all Internet traffic was the result of P2P file-sharing platforms\(^1\). P2P is the single largest consumer application of bandwidth on networks. P2P traffic significantly outweighs Web traffic and still continues to grow [1]. However, aside from a few one-off applications, P2P has not made a very large impact in systems design. The problem that P2P addresses is how to build reliable, large-scale systems without a high cost or management load on any one individual or organization. This is a very appealing vision. Many have the desire to distribute to a large audience without having to pay proportional to the number of people that receive the media. Additionally, many small- and medium-sized organizations need computing infrastructures, such as network file storage or job scheduling, but do not have the resources to pay a full-time system administrator. P2P design approaches should dramatically increase the reach of distributed computing systems by keeping costs incredibly low.

P2P has not yet delivered on its promise because, in most cases, to deploy a P2P-based system developers must start almost from scratch, or work with very simple primitives, such as a distributed hash table (DHT) [2–5]. The start-from-nothing approach is used by such applications as Gnutella [6], Bittorrent [7], and Skype [8], which are among the few widely used P2P systems. In contrast, to develop client-server

\(^1\) According to http://www.ipoque.com/
based applications developers, have access to the Berkeley sockets library [9] and
Web frameworks such as those based on Java [10], Python-based Django [11] or
Ruby-based Rails [12]. Such frameworks and abstractions make it easy to deploy large
client-server systems without having to start from scratch, i.e., sending Internet protocol
(IP) packets between hosts like most P2P applications do.

This dissertation enables a new generation of P2P systems by developing new P2P
abstractions and search frameworks that free developers from the problem of starting
from nothing to build each application. I deliver new research results about distributed
data structures as well as new search paradigms beyond DHTs. The research described
by this dissertation addresses barriers that are preventing the wide-scale adoption of
P2P by systems designers.

By the nature of P2P networks, each node does not maintain global information,
such as the number of peers in a network. In P2P networks, keeping track of network
size is often a challenging problem because the nodes are distributed, and thus, typically
know only a subset of the nodes in the system. However, there are many applications
where each node needs to know the size of P2P network, at least approximately.
System monitoring and obtaining global statistics becomes much more complex. I
propose four network size-estimation algorithms. The accuracy of a network size
algorithm has an great impact on several applications on P2P networks including
data retrieval. The methods utilize node density over various lengths in a network.
The node density makes one estimate the number of nodes in the whole system by
\( \hat{N} = LD \), where \( \hat{N} \), \( L \), and \( D \) represent the estimated network size, the length of address
space, and the node density, respectively. Each estimation method requires different
communication costs and achieves different performance. Thus, a user can select one
of the methods with respect to his purpose and expected performance.

Early P2P applications used simple broadcasting method for their query systems.
In systems like Gnutella, nodes connect to a subset of neighbors and send queries to all
neighbors. To implement such flooding-based query systems is simple, but the flooding is costly in that a query message should be delivered to almost all nodes in the system. In contrast, Distributed Hash-table (DHT) systems are very efficient for key-based lookup. One of the downsides of DHT systems is the difficulty of implementation, because they map the data structure of a hash table onto the structure of the P2P network. Also, DHT-based systems do not have a good solution for general queries because it is hard to map many interesting queries onto the DHT. For example, regular expressions cannot be mapped to metric space.

In this dissertation I present Deetoo, an efficient query-resolution algorithm. The heart of this work is simple data object replication and a search algorithm. In this model, the usual 1-D overlay topology is transformed into a rectangular matrix. Replication and query resolution are executed by bounded broadcasting over the columns and rows on the matrix space, respectively. Deetoo’s probabilistic approach achieves a constant success rate. Distributed systems should avoid massive communication cost. Deetoo stores $O(\sqrt{N})$ replicas per object with a high probability. Unlike the structured network, Deetoo does not require that each replicated object be mapped into a DHT. A query generates $O(\sqrt{N})$ messages. This implies more messages when compared to DHTs, which have only $O(\log N)$ complexity, but offers a major improvement over flooding-based searches with $O(N)$. The storage cost associated with the $O(\sqrt{N})$ size of the replica set in Deetoo is acceptable, especially when handling metadata instead of large multimedia files. Deetoo is the first system in which arbitrary unstructured queries can be mapped to a structured P2P system with query cost $O(\sqrt{N})$. Deetoo’s searching algorithm can coexist on the same P2P network with a DHT, so with Deetoo users can still have structured as well as unstructured queries. Also, users can control success rate and search cost with Deetoo by adjusting the degree of replication with a user-specified value. Our protocol search time outperforms other unstructured models by forming an efficient local tree: Deetoo completes searches in $O(\log^2 N)$ time by using
Kleinberg’s small-world network model. Lastly, due to the fact that Deetoo is built on a structured overlay, objects can be updated or deleted after being published on the system. This represents an advantage over random walk based schemes previously proposed in the literature.

Deetoo is a scalable, self-organizing, and flexible unstructured search system on top of a structured overlay. Users can control a hit rate by increasing or decreasing a replication factor. However, Deetoo is not suitable for systems requiring a complete search resolution, such as distributed P2P database systems, since virtually the communication cost increases up to $O(N)$. Exact Deetoo is designed to address the problem of Deetoo’s inability to achieve 100% hits without paying higher communication cost than Deetoo’s. The design of Exact Deetoo enables a collection of address bins to be occupied by at least one node. The collection of address bins is called a box. By maintaining the occupancy of each box, a querying node always finds a match from the nodes in a box no matter where it initiates a query. While Deetoo’s join cost remains very small, Exact Deetoo requires higher join cost. However, once nodes are joined a network, the message overhead required for either a caching or a querying message is $\sqrt{N}$. 
2.1 Network Size Estimation on Structured P2P Systems

The simple solution for estimating network size is that the requesting peer sends a simple broadcasting message to all the nodes in the system. The benefit from this method is that it guarantees the accuracy of the network size and it can be applied any type of network topology from random to structured. However, the method is impractical in real systems due to network dynamics as well as the high message overhead for very large networks ($O(N)$ cost complexity). In addition, the time to acquire the network size information is too high. To address the problems of broadcasting over an entire network, several approaches have been proposed to estimate the size of P2P networks, which are either random or structured.

For the random P2P topology, several gossip-based estimation methods have been proposed. Kempe et al. [13] suggested a push-based gossip aggregation scheme. The suggested method requires to keep track of a normalization factor and this results in a complicated update and exchange process. The similar push-based gossiping was introduced by Jelasity et al. [14]. In this method, one node initializes its counter value to 1, while other nodes set it to 0 initially. Then the counter value is updated to the mean of its own previous value and the previous value from the communicating node when a pair of nodes communicates. After massive message changes between nodes, all the nodes share the same counter value. The method requires $O(N^{1+2/d})$ of message overheads for a d-dimensional random graph and $O(N \log N)$ for expander graphs. However, this method is not suitable for highly dynamic networks where underlying directed graph is not strongly connected, as this algorithm is highly sensitive to node failures in the early stage of the algorithm. Massoulié et al. [15] proposed two measurement techniques that are applicable to arbitrary overlay networks. The first method, called the Random Tour method, initiates a message that executes a random walk until it finds the initiating
node. They used a continuous time random walk, which yields unbiased samples instead of a discrete time random walk. The Random Tour fits for small networks since cost increases linearly as a network grows. The second method is called Sample and Collide. The method utilizes “Inverted Birthday Paradox” [16]. The methods proposed in both papers select uniform random samples, and then uses such random samples to produce an estimate of system size, based on how many random samples are required before two samples return the same peer. The cost of [16] is $l\sqrt{N}$, where $N$ is the number of nodes in the system and $l$ is some real number, and for a target relative error of $1/\sqrt{N}$. [15] improves the cost to $\sqrt{lN}$ by selecting uniform random samples to achieve the same accuracy.

In structured P2P overlay networks, node identifiers are selected uniformly at random, which means the distances between a pair of nodes are the same all over the identifier space. Horowitz et al. [17] proposed the estimation method by calculating the density of identifiers in a ring-based structure. Their method costs less since the initiating node only needs to communicate with its neighbor nodes. But the resulting estimation is not accurate, the expected accuracy being in the range $[n/2, n]$. Mahajan et al. [18] have a similar idea using the density of node identifiers in Pastry’s leaf-set, but they do not provide analytical or simulation results. Similar works are found in [14, 19, 20]. In [21–23], a spanning tree from an overlay network is generated to estimate the size of the network. The estimated values are aggregated along the tree. The method using a spanning tree takes $\Theta(N)$ of communication cost. While the method is accurate, they did not consider the network dynamics. An interesting approach is proposed in [24]. They estimate the network size by observing node degrees. The assumption for the method is a power-law structure for the distribution of node degree. Bawa et al. [16] also introduced a similar approach observing node degrees on top of Erdős-Rényi random graph. The cost of the latter method is $O(\log N)$. 
2.2 Search in P2P Systems

Search methods for P2P systems can be categorized into two broad systems: DHT-based structured look-ups and unstructured searches using flooding or random walks. Highly structured P2P networks with DHT look-up algorithms [2–5] are efficient in that these networks achieve low query costs of $\log N$. This is because they place data objects at particular points on the network topology which are determined by an object’s key. Beehive [25] achieves constant look-up latency on top of a DHT through proactive replication. Despite their efficiency, the possibility of operation, even with extremely unreliable nodes, has not been yet examined. In addition, it is impossible for Beehive to handle high-dimensional complex queries. Extensive research has been conducted to address the limits of the exact search problems in DHT. One example is pSearch [26]. A rolling index reduces the number of dimensions for mapping purposes onto the overlay and divides the semantic vector (SV) into sub-vectors. Each sub-vector has the same number of dimensions as the CAN overlay does. Although pSearch is simple and supports high-dimensional queries, it requires control on the data objects of each node. Especially, when nodes join and leave frequently, pSearch incurs massive overhead. Therefore, it is more suitable for networks with stable nodes rather than for highly dynamic networks. pSearch still requires mapping search index into structured P2P overlay, and this limits the support of general query. Cubit [27] provides keyword search capability over a DHT. Cubit efficiently finds multiple closest data sets for a given query. However, it requires the creation of a keyword metric space and only returns multiple similar results to compensate for typos in queries.

More complex query resolution methods have been explored for P2P resource discovery for grids. SWORD [28], Mercury [29], and MAAN [30] are proposed to support multi-attribute range queries on top of structured overlays. In SWORD and MAAN, a DHT is created for each attribute. The number of created DHTs is the same as the number of attributes. Mercury also maintains a logical overlay for each attribute.
but it does not use DHTs. For few parameters and very narrow range queries, these systems can outperform Deetoo. Otherwise, for large numbers of parameters and larger query ranges, Deetoo can perform better as it always requires the same $O(\sqrt{N})$ complexity regardless of query or data type. Another drawback of DHT-based range query systems is that maintaining multiple overlays costs network traffic because update traffic increases as the number of attributes are increased. Deetoo creates two overlays and update traffic takes place only in the caching overlay. Moreover, because data is not structured, Deetoo’s query is not limited to range query. Since the data objects as well as query messages need to be mapped onto network structure in DHT-based systems, there is no easy way for DHT systems to support complex queries such as multi-dimensional queries. Deetoo does not demand any keyword mapping into DHTs and can execute more general queries like regular expression searches.

Unlike DHTs, unstructured P2P systems mostly depend on flooding for message transfers. The big advantage of flooding-based P2P systems is the capacity for high-dimensional search. An early version of Gnutella was based on naive flooding. Because flooding produces a very large number of messages over an entire network, pure flooding limits network size. To address this scaling problem, various types of solutions have been proposed. KaZaa [31] and iXChange [32] introduced central server-like super-peers. However, super-peers cause bottlenecks, security issues, and single point of failure problems due to their server-like characteristics. Recently, research efforts have also focused on locality-based flooding. Systems adopting interest-based locality [33–35] assume that two peers having common interests share pieces of a data object. Under this assumption, a shortcut connection is established between two peers having common interests, and queries from one peer are delivered to the other through this shortcut link in the first stage of flooding. Locality-based flooding requires warm-up procedures to gather query history for shortcut connections. LightFlood [36] uses a neighbor-degree-based locality scheme. LightFlood forms a
tree-like sub-overlay called FloodNet using neighbors’ degree information. Once the
sub-overlay is formed, there are two overlay networks in the system: the original P2P
overlay and FloodNet. The flooding takes two stages. Messages are transmitted using
pure flooding with relatively small TTL values in the first stage. The peers that receive
the query with zero TTL trigger the second stage of flooding in the FloodNet. Although
LightFlood is simple and helps stop generating massive messages for queries at a
certain point, searching unpopular objects requires the entire network to be visited; in
addition, LightFlood needs to be warmed up.

A random walk-based search technique is introduced by Adamic et al. in [37].
Their work reduces search cost by the factor of the number of replicas, but they do
not consider replica placement. Although random walk searches have an advantage
over flooding in terms of search cost, they has some scalability problems because
almost all the queries tend to concentrate on the high-degree nodes. To address
this problem, object replication using the square-root principle [38, 39] and topology
reconstruction [40] have been proposed. Both reduce search time but incur considerable
communication cost to maintain fresh topologies or data replication copies. Popularity-biased
random walk [41] achieves the square-root principle without the cost of data movement
or topology reconstruction. Sarshar et al. [42] combined flooding and random walking in
power-law networks. In their work, a query can be resolved in time \(O(\log N)\). However,
\(O(N \times \frac{2\log k_{\text{max}}}{k_{\text{max}}})\) messages are transmitted for a single query, where \(k_{\text{max}}\) denotes the
maximum degree; thus, when \(k_{\text{max}} = \sqrt{N}\) this becomes \(O(\sqrt{N} \log N)\).

Liu et al. [43] studied bounded broadcasting in wireless sensor networks. A
balanced push and pull strategy achieves \(O(\sqrt{N})\) search cost in the best scenario.
However, comb-needle data discovery technique requires nodes to estimate cache and
query frequency. All nodes in the network should keep their location information in the
grid. Though they do not analyze search time, it is possible to estimate it. By the nature
of the hop-by-hop message transfer, the comb-needle data discovery takes linear time
in the bounded range which are $O(\sqrt{N})$. The dynamic paths quorum system [44] is scalable and operates in a dynamic setting. The quorum sets are divided into reading quorums and writing quorums in a grid, and each reading quorum intersects each writing quorum. They analyzed probe complexity and availability, especially in a dynamic environment. The probe complexities of the non-adaptive (without stabilization) and adaptive (with stabilization) algorithms is $\Theta(\sqrt{N \log N})$ and $\Theta(\sqrt{N})$, respectively.
CHAPTER 3
NETWORK SIZE ESTIMATION IN PEER-TO-PEER NETWORKS

3.1 Introduction

Since Peer-to-Peer systems are decentralized, each node does not maintain global information, such as the number of peers in a network. In P2P networks, keeping track of network size is often a challenging problem because each node typically knows only a subset of the nodes in the system. However, there are many applications for which each node needs to know the size of the P2P network, at least approximately. The system monitoring and obtaining global statistics become much more complex. For instance, data replication applications largely depend on the number of nodes in determining the replication rate of data objects and queries. The number of nodes is used for load balancing as well as for limiting broadcasts. The accuracy of network size information is often very important to these applications. In search applications, both underestimation and overestimation affect performance in success probability and communication cost. If a node underestimates the size of the network, the success probability goes down below average while the communication cost is reduced. The exact opposite result is expected if a bigger size is estimated.

The simple way to count the number of nodes in a system is to broadcast a simple message to all the nodes in the network. However, this method generates high communication cost in that all nodes in the system must be visited, and it requires extremely large time overheads. Thus, it is not suitable for large-scale networks. Moreover, I can not count on the results of broadcasting in a complete network may not apply in P2P networks, considering that nodes in P2P networks join and leave frequently. The main purpose of this dissertation is to obtain a good estimation without paying too much communication and time cost.

In this dissertation, I propose four different network size estimation methods using only local information to reduce network size estimation error. All the proposed
approaches are based on the assumption that all nodes are uniformly distributed in an address space. In other words, the expected distance between any two nodes is equal. Based on this assumption, I estimate the number of nodes by calculating the node density within a given length using the algorithm. Let $D$ be the node density and $l$ and $i$ be the length and the number of nodes in the length. Then, a node density is obtained by $D = i/l$. Using node density, network size can be estimated by $\hat{N} = LD$, where $L$ is the length of address space, for example, $L = 2^{160}$ used in Chord. I also combine two or more methods to increase the accuracy of estimation. A proper algorithm should be applied according to its objectives and applications.

- The first method calculates an average distance between two nodes by looking at the left and the right neighbors in a ring.

- The second estimation depends on the first method’s estimation ($N_0$). The initiating node contacts a node which is $\log N_0$ hops away. The number of nodes between these two nodes is $\log N_0$ and the distance can be easily calculated by comparing the addresses. Communication cost for this method is $\log N_0$.

- The third methods collects the results of method 2 from a node’s shortcut neighbors. Then the initiating node takes the median of the results.

- The last method uses bounded broadcasting whose bound is $\sqrt{N}$.

Methods 2, 3, and 4 demand previous estimation. Any of the methods can be more optimal than the others under special circumstances, such as network model. For example, if nodes are perfectly distributed uniformly at random in address space, the first method is enough to measure the network size with the least amount of cost because distances between any two nodes are equal. However, in a practical network, nodes are not perfectly distributed uniformly at random in address space. In other words, the distances between two pairs of nodes are not the same in a practical system. Thus, as the size of sampling is increased, the accuracy of the network size estimation tends to be higher by calculating the average distance between two nodes. Increasing sampling size also causes higher communication cost.
Another approach to limit network size estimation errors is to make a network have more uniformly distributed peers by controlling joining method. When a new node joins a network, the approach provides two possible addresses and lets the node select the one that maximized the sum of distances to its adjacent nodes. As a result, the method makes distances between every two nodes more evenly distributed, though simply choosing a random address is much simpler.

I compare the accuracy and the communication cost using each method individually and combined methods in simulation.

3.2 Network Size Estimation in P2P Networks: Theoretical Background

In this section, four different methods for network size estimation are presented. Except the third method (median of shortcut neighbors’ estimation), each algorithm calculates a density in address space. A node density \(D\) can be obtained by \(n_{ab}/d_{ab}\), where \(n_{ab}\) is the number of nodes between node \(a\) and node \(b\), and \(d_{ab}\) is distance between node \(a\) and node \(b\). The node density \(D\) is used to estimate total number of nodes in a network by \(\hat{N} = DL\), where \(L\) is the length of address space, for example, \(L = 2^{160}\). As the number of samples increases, the accuracy of estimation also increases. Thus, experiments were conducted under various settings to get the best estimation of the network size without paying too much communication cost. If more nodes are involved in calculating the node density among the nodes, the level of accuracy goes up, though it requires more time and bandwidth. On the contrary, the variance of the estimated network size is much higher when only two nearest neighbors are used for the calculation. Accuracy of size estimation is lost in this case, though a savings in time and bandwidth is observed. Simulation results confirm that the variance of estimated network size goes lower when more nodes are involved in the estimation.

3.2.1 The Algorithm

The first algorithm, local estimation, does not require any message exchange between nodes. Since each peer keeps its neighbor peers’ information, such as
neighbor peers’ addresses, it can calculate node density using this information. Let \( d_r \) and \( d_l \) represent the distance to the right neighbor and the distance to the left neighbor, respectively. Then, node density is \( D = \frac{2}{d_r + d_l} \). Finally, the estimation is completed by \( N_0 = LD \). This method has a critical weakness even though no message overhead is generated. The variance of estimated network size from each node is very big because this algorithm counts on the uniformity of the node’s inter-node distances to its neighbor nodes. Thus, this method is utilized as the initial estimation for the rest of the methods. However, if there exists a network model whose inter-node distances are fairly uniformly distributed, this method is the best algorithm in that the result is accurate without message overheads.

The second method, log estimation, has more nodes involved in the node density calculation. The initiating node sends a ping message to its left (or right) neighbor with \( \log N_0 - 1 \) TTL, time-to-live, based on the initial estimation \( N_0 \). The TTL is decreased by 1 while the message is passed to the next left (or right) neighbor. The TTL reaches 0 in a \( \log N_0 \)-hop-away node from the initiating node. The node sends a message back with its address information to the initiating node. The initiating node calculates the distance (say, \( d_{\log} \)) to the \( \log N_0 \)-hop-away node based on the address. The node density is obtained by \( D = \frac{\log N}{d_{\log}} \).

The third method, median estimation, takes the median of the estimated network sizes using log estimation of the initiating node’s long-range connections. The nearest-neighbors estimation is excluded because the distribution of inter-node distances from adjacent peers already affects the calculation of node density from the initiating node. The estimated network sizes of the adjacent peers and the initiating node tend to be similar. Median is preferred to mean in this algorithm because median is a better measure of central tendencies that discounts the importance of numbers outside the data range.
The last method, square-root estimation, utilizes bounded broadcasting which limits the broadcasting to $\sqrt{f_{rep}N_0}$ nodes. Similar to the log-estimation, the initial estimation ($N_0$) is the result of local estimation and $f_{rep}$ is a replication factor. A bounded broadcast is accomplished with the following recursive algorithm: to broadcast a message over the region $[\alpha, \beta]$, the routing algorithm finds any node in the given range firstly via greedy routing, in which a node finds the closest node to the destination as its next node. Note that the number of nodes in the range $[\alpha, \beta]$ is $\sqrt{f_{rep}N_0}$. Let us assume that a node $\alpha$ is the first node in the range recognized by greedy routing, then the message is sent to node $\alpha$. Suppose $\alpha$ has $F$ connections to nodes in the range $[\alpha, \beta]$. The $i$th such neighbor is denoted as $b_i$. The node $\alpha$ sends a bounded broadcast over a sub-range, $[b_i, b_{i+1})$, to $b_i$, except the final neighbor. Differently stated, $b_i$ is in charge of bounded-broadcasting in the sub-range $[b_i, b_{i+1})$. If there is no connection to a node in the sub-range, the recursion is ended and the node stays in the tree as a leaf node. To the final neighbor ($b_F$), $\alpha$ sends a bounded broadcast to $[b_F, \beta]$. When a node receives a message to a range that contains its own address the message is delivered to that node in addition to being routed to others. Figure 3-1 shows how this bounded broadcast forms a local tree recursively in a range of $[\alpha, \beta]$. The time required for this is $O(\log^2 N)$. By Deetoo’s recursive bounded broadcasting algorithm, all nodes in the range are involved in forming the local tree because only a node with no connection in the sub-range to the node which is responsible for bounded-broadcasting can stop recursion out of each branch of the tree.

3.2.2 Node Joining Algorithm for More Evenly Distributed Network

Although I assumed that each node is distributed uniformly at random, in a real network, inter-node distance varies. Moreover, nodes in P2P networks join and leave frequently. It is impossible to maintain equally distributed inter-node distances even if the distribution of initial network is quite uniformly distributed. The high variance of inter-node distances has a negative influence on estimation of network size because
it is directly against the assumption of equal distribution. To compensate for the non-uniformity of inter-node distances, I introduce a new joining algorithm. When a new node joins a network, the following steps are taken:

1. select two different random addresses,
2. at each address, calculate minimum ring distances to neighboring nodes which have left closest address and right closest address,
3. find a proper place on the virtual ring space by selecting the address that has the larger minimum distance to a neighbor node,
4. obtain addresses for and make connections to two adjacent neighbors and one short-cut neighbor based on Kleinberg’s inverse $r^{th}$-power distribution,
5. as a final step, copy objects from neighbors in the same set of sub-rings (in the same columns in matrix space).

Figure 3-2 shows how a new node finds its right place. From randomly selected positions $A$ and $B$, the nearest neighbors are node1 and node3. Between these two nodes, node3 has a bigger distance to $A$ than the distance from node1 to $B$. Thus, a new node takes a position $A$ as its new address. Our joining algorithm makes inter-node distances more evenly distributed because it always minimizes the maximum inter-node distance.

### 3.2.3 Communication Cost

To evaluate communication cost is meaningful because it is an important factor of network scalability. The reason why a simple broadcasting method cannot be applied to a large network is that this method is impractical. The number of round trip messages cannot be regarded as the communication cost for simple analysis. Local estimation requires no message exchange because in the network model, each node maintains the neighbor nodes’ information, including their addresses. The distance to the neighbor node can be directly calculated by comparing the addresses. The cost for the log estimation is $\log N_0$. Because the cost is logarithmic of the initial estimation, the effect of the variance of the initial estimation is relatively small. The median estimation itself
requires $d_n - 2$ messages, where $d_n$ denotes the degree of a node $n$. However, because this algorithm uses the result of log estimation from the node’s shortcut neighbors, the overall cost is the sum of log estimation and median estimation, which is $d_n \times \log N_0$. The cost for square-root estimation is $\sqrt{f_{rep}N_0}$.

### 3.3 Simulation Results

In this section, the performance of network size estimation is evaluated via simulation. Simulation results from each method are compared in terms of estimated network size distribution and communication costs.

#### 3.3.1 Simulation Methodology

To evaluate the performance of the work, the algorithm is simulated using Netmodeler [45], which is a free software and open source library written in C++. The only algorithm to require a parameter is the square-root estimation. The replication factor, $f_{rep}$ is set to 1. All estimation processes are initiated on every nodes in the network regardless of the algorithm. For the square-root estimation, messages are bounded broadcasted within randomly chosen ranges. Each simulation is repeated in the network size of 5,000 nodes and 500,000 nodes.

#### 3.3.2 Network Topology

The network model for the work is based on Kleinberg’s one-dimensional construction [46]. In this model, nodes in a ring are organized such that each node has a set of "local" contacts and one "long-range" contact. A small world model is widely used for P2P systems because it allows a reduced path discovery with only local information by forming a local tree in the search space. In addition to the search efficiency, a small world model requires low maintenance cost.

#### 3.3.3 Simulation Results

In this section, the network sizes estimated by four methods described in the previous section are compared via simulation. In the topology model on top of Netmodeler, the algorithms in two network models are compared. The difference
between network models is the joining process. In the random address model, a newly joining node selects a random address and this address specifies the location of the node in a ring topology. In the more evenly distributed network model, a new node chooses its address from two randomly selected addresses as described in section 3.2.2. Estimation is repeated by all nodes in the network and averaged. The main metrics for evaluating the proposed algorithms include average estimated network size and communication cost. High accuracy of network size estimation can be achieved if more nodes are involved in the estimation. In the other hand, the involvement of more nodes causes higher communication cost. Simulation results show a vivid trade-off between the accuracy and the cost.

Table 3-1 shows the results of estimation algorithms when the random address selection joining algorithm is used, and the results of estimation with a new joining method for a more uniformly distributed network is shown in Table 3-2. The estimates obtained by nearest estimation deviate by a large amount, and the estimation error is the biggest among the methods. The results confirm that the accuracy increases as the number of nodes used in calculating the node density is increased. The standard deviation is reduced by 40 to 60% when more uniform address selection is used for the network’s joining algorithm.

The accuracy of estimation algorithms are compared in Figure 3-3 with network size of 5,000 and in Figure 3-4 with network size of 500,000. The differences among methods are more vivid when the standard deviations are compared as shown in Figure 3-5 and Figure 3-6.

Figure 3-7 and 3-8 show communication cost with network sizes of 5,000 and 500,000, respectively. In the figures, method 1 is the nearest estimation, method 2 is the log estimation, method 3 is the median estimation, and method 4 is the square-root estimation. The communication cost for method 2 and method 3 is calculated cumulatively. Since method 2 depends on the results of method 1, the cost
for method 2 is calculated by $\text{cost}(m1) \times \log N_0$. Similarly, the costs for method 1 and method 2 are added for the calculation of the cost of method 3. The communication cost of method 1, 2, and 3 does not vary. The cost for method 2 is decided by the initial estimation by method 1. The cost for method 1 and method 3 is decided by the degree of nodes in the network. The average degree of nodes was observed to be around 4 regardless of the network size. In other words, each node maintains two direct neighbors in a ring and 2 long-distance connections.

The average cost for square-root estimation between the two network models is close, but the variance of more uniform model is lower than that of the random model. The low variance of the initial estimation by method 1 explains the low variance of square-root estimation.

Figure 3-9 shows the measured results of Mean Square Error (MSE)-cost relationship using log estimation. To see the trade-off between the cost and the mean square error in log estimation, the test is repeated with different weights, namely $w$. A $w \log N$-away node is selected to calculate the density between the node and the requesting node. As shown in the figure, the bigger the weight, the smaller the error. However, there was no big improvement beyond the weight of 3.

### 3.4 Conclusion

I present four network size estimation algorithms. Except the median estimation method, all algorithms estimate the number of nodes in a network using the node density. Each method provides distinctive accuracy and cost. Thus, users can select one of the four estimation algorithms by an application's demand. The nearest estimation is suitable for overlays that are uniformly distributed in an address space. The log estimation is inspired by DHT-based systems. The median estimation performs better if a node has more shortcut neighbors. The square root estimation can piggy-back on Deetoo’s cache or query messages. By combining all or several of the above methods, a user can can achieve a combination of accuracy and cost.
To improve the accuracy of the estimation algorithms, a new node joining algorithm is provided which increases the uniformity of inter-node distances. With the help of new joining algorithm, lower variance of each estimation method is observed.

Finally, the cost and the accuracy of each proposed method are evaluated by simulations. The simulation results confirmed the trade-off between the cost and the accuracy.
Table 3-1. Statistics of estimated network size (random address selection).

<table>
<thead>
<tr>
<th>Actual Size</th>
<th>Method</th>
<th>Average</th>
<th>Std. deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>1</td>
<td>10182</td>
<td>20391.08</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4950</td>
<td>1820.45</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>4772</td>
<td>1210.91</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5000</td>
<td>643.77</td>
</tr>
<tr>
<td>500000</td>
<td>1</td>
<td>996912</td>
<td>2028568.64</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>498570</td>
<td>143811.02</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>486665</td>
<td>97687.96</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>499976</td>
<td>21300.36</td>
</tr>
</tbody>
</table>

Table 3-2. Statistics of estimated network size (even address selection).

<table>
<thead>
<tr>
<th>Actual Size</th>
<th>Method</th>
<th>Average</th>
<th>Std. deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>1</td>
<td>7287</td>
<td>7409.59</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4692</td>
<td>1132.86</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>4617</td>
<td>810.11</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4985</td>
<td>467.00</td>
</tr>
<tr>
<td>500000</td>
<td>1</td>
<td>752250</td>
<td>1213434.81</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>480644</td>
<td>95900.29</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>474566</td>
<td>65750.32</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>499986</td>
<td>14096.09</td>
</tr>
</tbody>
</table>

Figure 3-1. Tree generation by bounded broadcasting
Figure 3-2. Address selection of newly joined node

Figure 3-3. Mean estimation with actual size of 5000 nodes
Figure 3-4. Mean estimation with actual size of 500000 nodes

Figure 3-5. Standard deviation of each estimation (N=5000)
Figure 3-6. Standard deviation of each estimation (N=500,000)

Figure 3-7. Communication cost with 5000 nodes
Figure 3-8. Communication cost with 500000 nodes

Figure 3-9. Communication cost vs. MSE with log-estimation (actual network size: 50,000)
CHAPTER 4  
DEETOO: SCALABLE UNSTRUCTURED SEARCH BUILT ON A STRUCTURED OVERLAY

4.1 Introduction

P2P query systems may be categorized into three different architectures: centralized, decentralized/structured, and decentralized/unstructured [39]. First, centralized networks rely on an indexing server for updates. Early P2P systems such as Napster fall into this category. Within this architecture, each node sends a query not to another peer, but to the central indexing server, to find a node that matches the query. These centralized P2P networks are clearly not scalable (unless they replicate sufficient back-up servers) and they are not fundamentally able to avoid the single-point of failure problem.

Structured networks use an overlay topology that is structured but without central control. In distributed hash table (DHT) based P2P overlays [2–5], each object has a unique identification key that plays a role in placing an object into the network. A query can also pinpoint the object indexed by a key. DHT-based overlays are not only efficient but also partially resistant to failure of the nodes or links. The chief disadvantage of structured networks is that queries must be mapped onto the structure of the network. Since it is not straightforward (or perhaps not possible) to map some queries onto low-dimensional spaces such as those formed by common P2P systems, not all query systems can be easily implemented on structured networks. An example might be regular expression matching. Furthermore, additional steps need to be taken to deal with the problem of hot-spots in the network, which form when demand for specific objects increases suddenly.

Unstructured networks use flooding-based routing and random walks to locate a resource and to resolve a query. These systems do not require a unique key for each object. Therefore, any type of query can be easily mapped to an unstructured P2P system. Unstructured systems with key-word searching are prevalent in P2P file-sharing networks. While these systems are resilient to both node joins and failures due to their
simplicity, unstructured file-sharing networks have a scaling problem. As the network size grows, each participating node can become overloaded with a huge amount of query messages, which can scale to a complexity of $O(N)$ since all the nodes need to be visited for a single query.

In this dissertation I present Deetoo, an efficient query-resolution algorithm based upon Kleinberg’s one-dimensional construction [46]. The idea is to organize nodes in a ring in which each node has a set of “local” contacts and one “long-range” contact. A small-world model allows a reduced path discovery with only local information by forming a local tree in the search space. In addition to the search efficiency and low maintenance cost, I can reuse existing P2P code designed for a ring topology by adapting a one-dimensional small-world network model. The heart of the work is simple data object replication and a search algorithm. In this model, the usual 1-D overlay topology is transformed into a rectangular matrix as described in Section 4.2. Replication and query resolution are executed by bounded broadcasting over the columns and rows on the matrix space respectively.

This chapter evaluates the novel Deetoo algorithms from a theoretical standpoint and through simulation. I focus on the behavior of the search algorithms for each of the following metrics: successful searching probability (hit rate), communication cost (bandwidth consumption or number of generated messages), and search time (depth of the multicasting tree). Measures of the query accuracy are important because precise information retrieval is one of the main purposes of P2P networks. Distributed systems should avoid massive communication cost. Deetoo stores $O(\sqrt{N})$ replicas per object with a high probability. Unlike the structured network, Deetoo does not require that each replicated object be mapped into a DHT. A query generates $O(\sqrt{N})$ messages. This implies more messages when compared to DHTs, which have only $O(\log N)$ complexity, but offers a major improvement over flooding-based searches with $O(N)$. Assume that data can be cached on $C$ of any $N$ nodes. Since there is no structure assumed, and if
load is evenly balanced across the nodes, caching does not depend on the data being cached and I assume that \( C \) nodes are selected at random. Similarly, when I query, I can check \( Q \) of the \( N \) nodes, again at random. Thus the probability I miss the stored data is approximately:

\[
\left( 1 - \frac{C}{N} \right)^Q = \left( 1 - \frac{C}{N} \right)^{\frac{CQ}{N}} \approx \exp\left(-\frac{CQ}{N}\right)
\]

(4–1)

So, I need \( CQ = O(N) \) in order to have a constant probability of success in this model. This presents an intuitive trade-off, the more nodes on which I cache data, the fewer I need to check to find it. Let \( p_c \) be a fraction of nodes that has caching, \( p_q \) be a fraction of nodes that has querying, \( \alpha \) be a replication factor, and \( K \) be a total cost both for the cache and the query. Note that \( p_c + p_q = 1 \). The replication factor is set to \( \alpha = \frac{CQ}{N} \).

Then,

\[
K = p_c C + p_q Q
\]

(4–2)

\[
= p_c \frac{\alpha \alpha N}{Q} + p_q Q
\]

(4–3)

By solving the equation for the minimum \( K \), the minimum cost for query is:

\[
Q = \sqrt{\frac{p_c}{p_q} \frac{\alpha N}{Q}}
\]

(4–4)

Similarly,

\[
C = \sqrt{\frac{p_q}{p_c} \frac{\alpha N}{Q}}
\]

(4–5)

To minimize \( C + Q \), which is the cost to cache an object and query for it once, the best choice is for \( C \) and \( Q \) to be \( O(\sqrt{N}) \) when \( p_c = p_q \). This simple analysis suggests that any unstructured load-balanced system will be require at least this much communication. Of course, logarithmic complexity would be nice, but for many practical systems, the number of nodes might well be 100 to 10,000 which mean \( C \) and \( Q \) values on the order of 10 to 100. Such overhead costs would be completely acceptable for many applications, especially when I handle meta data instead of large multimedia
files. Deetoo is the first system that arbitrary unstructured queries can be mapped to a structured P2P system with query cost $O(\sqrt{N})$. Our searching algorithm can coexist on the same P2P network with a DHT, so with Deetoo users can still have structured as well as unstructured queries. Also, users can control success rate and search cost with Deetoo by adjusting the degree of replication with a user-specified value. In [39], Lv et al. showed that square-root replication distribution is theoretically optimal in terms of minimizing the overall search traffic. Our protocol search time outperforms other unstructured models by forming an efficient local tree: Deetoo completes searches in $O(\log^2 N)$ time by using Kleinberg’s small-world network model. Lastly, due to the fact that Deetoo is built on a structured overlay, objects can be updated or deleted after being published on the system. This represents an advantage over random walk based schemes previously proposed in the literature.

4.2 The Deetoo Search Algorithm

In this section I describe the data structure and search algorithm I use for Deetoo. I take a similar approach to the idea of DHT P2P networks: take the hash table data structure, and build a distributed data version of this data structure where memory locations spread across many nodes. To understand the Deetoo P2P system, I will first describe a local data structure and then describe a distributed version of that data structure.

4.2.1 An Unstructured “Hash” Table

Consider a table data structure that has $B$ bins arranged in a $k \times l$ array ($B = kl$). I can say $b_{ij}$ is the bin in row $i$ column $j$. To add an object into this table, select a random column and insert the object into each bin in that column. Which is to say, choose a random value $r \in (1, l)$, and insert the object in the set of bins $C_r = \{b_{ir}|i \in (1, k)\}$. To search for an object, select a random row and check each bin in that row for the object. Equivalently, choose a random value $p \in (1, k)$ and look for the object in the set of bins $Q_p = \{b_{pj}|j \in (1, l)\}$. Since every row and column intersect at exactly one bin,
$C_r \cap Q_p = \{b_{pr}\}$, a query will always find one bin into which an object was inserted. The number of bin accesses to insert an object is $k$. The number of bin accesses to query for an object is $l$. A trade-off between cost of insertion and cost of searching exists.

As a local data structure, the above may have little value: it costs $k$ times more to store than an unsorted list, and the total number of comparisons needed for a search is still $M$ if there are $M$ objects in the table. However, as a distributed data structure designed to distribute load and minimize communication, it is useful since only $l$ bins need to be searched. This data structure achieves totally balanced load distribution in the network because each object is replicated over a bounded region irrespective of its popularity. On current unstructured systems, queries can concentrated on a specific peer having many objects. Besides, popular objects are likely to be available at several nodes and the probability of succeeding in queries for it is much higher. In other words, queries for rare objects are barely answered. Because Deetoo algorithm distributes objects evenly over the network, I can avoid creating hot-spots and improve probability of finding rare objects. In the next section I describe how to make a distributed version of this data structure which can support general data objects and queries. I will describe a randomized version of the above where queries succeed with a high probability.

### 4.2.2 A Distributed Unstructured Table

Our setting will be the standard P2P setting: there are $N$ nodes which can communicate, store objects and perform queries. In addition to nodes, there are also bins. Bins may or may not be occupied by a particular node. The number of bins is assumed to be fixed at some very large number, for instance $2^{160}$. The bins are arranged in a rectangular array, which in Deetoo I will assume to be square. In the data structure of Section 4.2.1, each row and column intersect at exactly one bin that may or may not be occupied by a node. Instead of querying and inserting along one row and one column, I will do so over a sufficient number of rows and columns such that the probability of having more than one node in the overlapping set is very likely to be one.
Figure 4-1 depicts the 2 – D array I use to cache (insert) and query (search). In Figure 4-1, the area $A$ represents the intersection of a particular cache and query operation. I see in that area, for example, there are three nodes, and so the query will be successful. I need to show several things to see that this algorithm will work in a distributed setting:

1. Show how to efficiently send cache and query messages to entire columns and rows respectively.

2. Show how to deal with nodes joining and leaving the network.

To deal with the above two problems I leverage existing work on building routable 1 – D P2P networks such as Chord[3] and the small-world model[46]. Rather than build a P2P network that is explicitly two dimensional, I build two sub-graphs, each of which is a 1-D ring, which I can call the query ring and the cache ring. Each P2P node has exactly one address, and hence position, on each ring. This is depicted in Figures 4-2, 4-3, and 4-4; these drawings illustrate how one-dimensional rings for querying and caching locally overlay atop of the 2-D Deetoo array. A node is cached at location $i$ with probability $c_i$ and a query node is placed at location $i$ with probability $q_i$ in the query ring. $N(x)$ and $N(x')$ denote nodes whose addresses are $x$ and $x'$, respectively. $x'$ is transposed address of $x$.

In order to achieve the effect of ordering the cache ring by increasing along the columns and the query ring to increase along the rows, the bin address for a given node in the query ring must be the transpose of the address on the cache ring, as depicted in Figures 4-2 and 4-3. Assume that the size of the address space, and hence number of bins, is $B = m^2$. The address mapping algorithm is as follows: Let $x$ denote an address on the cache ring. The address $x$ can be expressed by column element, $x_i$, and row element, $x_j$, such that $x = m x_i + x_j$. A translated address on the query ring $x'$ is obtained by exchanging column element and row element: $x' = m x_j + x_i$. 

43
So, each P2P node then has two addresses in virtual 1-D rings and follows the usual procedure for joining each of the two rings as described in 4.2.5. Notice that on the cache ring, the nodes in the same columns (and adjacent columns) have adjacent addresses. On the query ring, nodes in the same row (and adjacent rows) have adjacent addresses. Because the rings are efficiently routable, it is also efficient to send a message to a randomly selected node near the start of a column or row. Similarly, to reach all elements of a row or column, I can use a bounded broadcast on one of the two rings.

### 4.2.3 Bounded Broadcast

In Deetoo, a bounded broadcast is accomplished with the following recursive algorithm: To broadcast a message over the region \([\alpha, \beta]\), the routing algorithm finds any node in the given range firstly via greedy routing, in which a node finds the closest node to the destination as its next node. Let us assume that a node \(\alpha\) is the first node in the range recognized by greedy routing, then the message is sent to node \(\alpha\). Suppose \(\alpha\) has \(F\) connections to nodes in the range \([\alpha, \beta]\). I denote the \(i^{th}\) such neighbor as \(b_i\). The node \(\alpha\) sends a bounded broadcast over a sub-range, \([b_i, b_{i+1})\), to \(b_i\), except the final neighbor. Differently stated, \(b_i\) is in charge of bounded-broadcasting in the sub-range \([b_i, b_{i+1})\). If there is no connection to a node in the sub-range, the recursion is ended and the node stays in the tree as a leaf node. To the final neighbor \((b_F)\), \(\alpha\), sends a bounded broadcast to \([b_F, \beta]\). When a node receives a message to a range that contains its own address the message is delivered to that node in addition to being routed to others. Figure 4-5 shows how this bounded broadcast forms a local tree recursively in a range of \([\alpha, \beta]\). The time required for the bounded broadcasting is \(O(\log^2 N)\) as shown in Section 4.3.2. By Deetoo’s recursive bounded broadcasting algorithm, all nodes in the range are involved in forming the local tree because only a node with no connection in the sub-range to the node which is responsible for bounded-broadcasting can stop recursion out of each branch of the tree.
4.2.4 Data Insertion and Query

Deetoo is a search algorithm on a specific network structure. When an object is inserted at a node \( a \) as in Figure 4-6, Deetoo selects, at random, a range in which a query message is to be broadcasted locally. Node \( a \) starts greedy routing to find a node \( n \) in this random range and \( n \) starts bounded broadcasting within a limited range \((C)\) on the caching ring to replicate the object (Figure 4-7). Bounded broadcasting size for caching \((C)\) is given by \( C = \alpha \sqrt{\frac{B}{N}} \), where \( \alpha \), a replication factor, is a constant (details about this formulation is followed in Section 4.3). All nodes in the range of bounded broadcasting (all nodes in the shaded area in Figure 4-8) eventually receive a copy of an object, \( o \).

Query resolution follows the same bounded broadcasting steps. The only difference is that the query resolution is executed in the querying space in which every node’s address is transposed (stated in Section 4.2.2) in the caching space. Bounded broadcasting size for a query, \( Q \), follows the same formulation as the caching size \( C \) as shown above. Figures 4-9, 4-10, and 4-11 show the steps involved in a query: node \( a' \) issues a query for an object \( o \), node \( n' \) initiates a bounded broadcast within \( Q \), and node \( n(o) \) resolves a query, finally, \( n' \) retrieves an object from a node \( n(o) \). Since all nodes in the area \( A \) have a copy of an object \( o \), \( n' \) can retrieve a desired object from any of the nodes in \( A \). By following these steps, a query can be resolved if at least one node exists in \( A \). Note that a user can select bounded broadcasting size by adjusting the replication factor, \( \alpha \). The bigger the \( \alpha \), the higher the probability to hit a node in \( A \), at the expense of larger number of messages and replicas.

Users may want to insert objects into a network, or delete them from a network. The following explains how Deetoo manages both insertion and deletion.

- Object Insertion: When a new object is inserted to the network, copies of the object are created along some sets of columns in the matrix space (the number of columns is inversely proportional to the square-root of the network size) using bounded broadcasting. Unlike a DHT, each object does not have to have a unique
identification number, and each inserted object does not have to map a key to the "closest" node.

- Object Deletion: Objects do not stay forever in the networks. When an object is obsolete, Deetoo provides a deletion that uses both caching and querying. First, a query for an object which needs to be deleted is performed in order to retrieve the object's range. Note that each object writes its range information in itself or its metadata when it is cached. After the object's range is acquired, the 'DELETE' message is broadcasted within the range in the same way the object was cached. As a result, Deetoo guarantees all replicated objects that are supposed to be stored in the nodes in the range to be removed.

4.2.5 Node Joins, Leaves, and Stabilization

In this section, I describe how a node joins and leaves the network. Deetoo requires that each data object is cached over the desired range in a network. However, a new node joins the network with no cached objects. Success probability must stay constant regardless of node joins or leaves. Thus, it is necessary that all objects be replicated if a new node's address falls in a object's cached range. Deetoo relies on a new node copying objects from its new neighbors. I define this replicating process as stabilization in this chapter.

Each object must keep its range information as well as the object itself or meta data for the object where it was initially inserted to the network so that as nodes join and leave the objects can be maintained on their randomly selected column. After a node obtains a proper address successfully, the node sends requests to its newly connected neighbors in order to replicate objects in the network. Stabilization plays a role in this stage of replication. The following describes how all objects in the same boundary are copied to the new successfully joined node from neighbors.

1. Retrieve the object’s range from the neighbor’s object list.
2. Recalculate range with a given replication factor.
3. Copy the object from the neighbor node to the newly joined node only if the new node’s address lies on the recalculated range.
4. Repeat 1 through 3 for all objects from both adjacent neighbors.
If I assume that there exist $m$ unique objects in the network whose size is $N$, each node maintains $m \cdot \frac{\alpha}{\sqrt{N}}$ objects on average, where $\alpha$ is a replication factor. In the stage of stabilization, the node is able to receive objects from both neighbors. The maximum number of objects which need to be transferred is limited to $O\left(\frac{1}{\sqrt{N}}\right)$. I analyze stabilization cost in Section 4.3.3. I also simulate stabilization cost at the time of new node joins under churn and compare the simulation result with an analytical model. The simulation result shows that Deetoo requires very low stabilization cost. When a node leaves or fails, the protocol does not need to do anything because all objects have already been copied to all nodes in the same set of columns. Thus, neighbor nodes keep exactly the copies of the objects held by leaving or failing nodes.

4.2.6 Estimating Network Size

Each node in a Deetoo network is distributed without any server-client interaction, which makes it impossible for the peers to know exactly how many other peers exist. However, each node is required to know the network size in certain processes. For example, I cannot decide a caching or a querying size without knowing the number of peers in the network. The accuracy of network size estimation is key to deciding the optimal caching and querying range. Both underestimation and overestimation affect performance in success probability and communication cost. If a node underestimates the size of the network, the success probability goes down below average while the communication cost is reduced. The exact opposite result is expected if a bigger size is estimated. In the Deetoo simulation, the algorithms described in Chapter 3 are used. I experimented under various settings to get the best estimation of the network size. If more nodes are involved in calculating the average distance between two nodes, the level of accuracy goes up though it requires more time and bandwidth. On the contrary, the variance of the average distance is much higher when only two direct neighbors are used for the calculation. I lose accuracy of size estimation in this case though I save time and bandwidth.
I observed that the calculation of the average distance using $\log N$ hops is acceptable considering both accuracy and cost. Figure 4-12 shows estimated network size distribution under my network size estimating algorithm. In the simulation, I set the actual network size to 1,000. The average estimated network sizes are 1430.97 and 1695.95 for caching and querying, respectively. The standard deviations are 1248.56 and 1630.41, respectively. The medians are 1077 and 1144, respectively. Note that the error margin of overestimation is much larger than that of underestimation, which causes the average to be slightly larger than the actual size of network. Network size estimation for cache is more accurate than that for query, because new node’s address selection is based on caching address. The success probabilities depend on the distribution of network size estimation. In Figure 4-18, the simulation results show that the network size estimation works well in that the success probability is very close to the theoretical expectation. Success probability decreases slightly as network size grows, while expected success probability stays constant. This is because bigger networks have bigger error margin on estimation.

4.3 Analysis of Deetoo

Our analysis focuses on the probability of a query being successfully resolved, $P_s$, and the communication cost, $K$, to show that the Deetoo protocol for unstructured P2P networks is efficient and scalable. I build the following assumptions to simplify my analytical modeling. First, I have a fixed size of the addresses. Thus address space, $B$, is fixed, which is $m \times m$ for cache and query respectively as shown in Figure 4-1. Second, I assume that the caching probability I cache at location $i$, $c_i$, and the querying probability I put a query at location $j$, $q_j$, are both uniformly distributed. Third, I assume that the caching and querying probability are the same at any location. Finally, at most one node per a bin in matrix space is allowed for all nodes to have a unique address.
4.3.1 Exact Analysis for Success Probability

In Deetoo, queries are unstructured and load-balanced, which means that the probability that a given node receives a query is independent of the query and equal for all nodes. For each object and query pair, there is a region of size $A$ in the grid address space that receives both the query and the cache. The only way the query will not be resolved is that the region has no node. Since every configuration of $N$ nodes in the address space of size $B$ is equally likely, the probability there is a miss is calculated as following: First, I provide an exact analysis for miss probability. Assume that there are $N$ bins with $C$ being occupied, and I search $Q$ to try to find an occupied bin. The probability I fail to search is calculated as following:

- All $C$ occupied bins must be disjoint from the $Q$ I check and the number of possible combination is $\binom{N-Q}{C}$. Note that if $Q + C \geq N$, $\binom{N-Q}{C} = 0$
- The number of ways to arrange occupied bins is $\binom{N}{C}$

Thus, the probability of miss hit is:

$$P_{\text{miss}} = \frac{\binom{N-Q}{C}}{\binom{N}{C}} \quad (4-6)$$

$$= \frac{(N-Q)!}{N!} \cdot \frac{(N-C)!}{(N-C-Q)!} \quad (4-7)$$

There is four possible cases for the combination of the size of $C$ and $Q$. I analyze the upper bound and the lower bound for each case.

- **case 0**: If $Q = 0$ and $C = 0$, $P_{\text{miss}} = 1$.
- **case 1**: If $Q + C > N$, $P_{\text{miss}} = 0$.
- **case 2**: If $Q + C = N$, $P_{\text{miss}} = \frac{1}{\binom{N}{C}}$. since $N \leq \binom{N}{C} \leq 2^{-N}$ for $C \geq 1$,

$$2^{-N} \leq P_{\text{miss}} \leq \frac{1}{N} \quad (4-8)$$
• case 3: Q+C+E=N, where E ≥ 1. In this case, at least one bin is not occupied or searched. I use Sterling’s bounds to analyze the bounds for this case:

$$e^{-\frac{1}{12(n+1)}} < \frac{n!}{\sqrt{2\pi}n^n e^{-n}} < e^{-\frac{1}{12n}}$$

(4–9)

The cases are fairly trivial to evaluate except the sparse case (case 3), which is to say the query/cache overlap region and the number of nodes are both small compared to the total number of bins E can be thought of as the number of empty bins in the case of a miss. Look at the upper bound first.

$$P_{\text{miss}} < \frac{\sqrt{N-Q} \cdot N_Q \cdot e^{-(N-Q)} \cdot e^{-\frac{1}{12(N-Q)}}}{\sqrt{N} \cdot N \cdot e^{-\frac{1}{12N}}} \cdot \sqrt{N-C-Q} \cdot (N-C) \cdot e^P \cdot (N-C) e^{-\frac{1}{12(N-C)}}$$

$$= \frac{\sqrt{(N-Q) \cdot (N-C)}}{N(N-C-Q)} \cdot \left( \frac{(N-Q) \cdot (N-C)}{N(N-C-Q)} \right)^N \cdot \left( \frac{N-C-Q}{N-Q} \right)^Q \cdot \left( \frac{N-C-Q}{N-C} \right)^C e^{\gamma}$$

$$= \left( \frac{(N-Q) \cdot (N-C)}{N(N-C-Q)} \right)^{N+\frac{1}{2}} \cdot \left( \frac{N-C-Q}{N-Q} \right)^Q \cdot \left( \frac{N-C-Q}{N-C} \right)^C e^{\gamma}$$

$$= \left( \frac{(N-Q) \cdot (N-C)}{N \cdot E} \right)^{E+Q+C+\frac{1}{2}} \cdot \left( \frac{E}{N-Q} \right)^Q \cdot \left( \frac{E}{N-C} \right)^C e^{\gamma}$$

Due to the convexity of \( \log(1 \pm x) \), we know that

$$\begin{align*}
(1-x)^\sigma &< e^{-\sigma x} \\
(1+x)^\sigma &< e^{\sigma x}
\end{align*}$$

(4–10)

(4–11)

Thus,

$$\begin{align*}
\left( \frac{N-C}{N} \right)^Q &= \left( 1 - \frac{C}{N} \right)^Q < e^{-CQ/N} \\
\left( \frac{N-Q}{N} \right)^C &= \left( 1 - \frac{Q}{N} \right)^C < e^{-CQ/N} \\
\left( 1 + \frac{QC}{NE} \right)^E &< e^{CQ/N}
\end{align*}$$

(4–12)

(4–13)

(4–14)
Finally,

\[ P_{\text{miss}} < \left( 1 + \frac{QC}{NE} \right)^{E+\frac{1}{2}} e^{-2QC\gamma} \tag{4-15} \]

\[ < e^{-\frac{QC}{N}} \sqrt{1 + \frac{QC}{NE}} e^\gamma \tag{4-16} \]

where

\[ e^\gamma = e^{-\frac{1}{12(N-Q)} - \frac{1}{12(N-C)} + \frac{1}{12N} + \frac{1}{12(N-C-Q)} + \frac{1}{12}} \tag{4-17} \]

\[ < e^{\frac{1}{12E+1}} \tag{4-18} \]

Thus if I set C and Q such that \( CQ = \alpha B/N \), I have \( P_{\text{miss}} < e^{-\alpha} \sqrt{1 + \frac{\alpha}{Ee^{\frac{1}{12E+1}}}} \).

One can find a similar lower bound:

\[ P_{\text{miss}} = \frac{\sqrt{N - Q(N - Q)^N} e^{-\frac{1}{12(N-Q)}} \sqrt{N - C(N - C)^N}}{\sqrt{NNe^{-\frac{1}{12N}} \sqrt{N - C - Q(N - C - Q)^N}}} \]

\[ = \sqrt{\frac{(N - Q)(N - C)}{N(N - C - Q)}} \left( \frac{(N - Q)(N - C)}{N(N - C - Q)} \right)^N \left( \frac{N - C - Q}{N - Q} \right)^Q \left( \frac{N - C - Q}{N - C} \right)^C e^\delta \]

\[ = \left( \frac{(N - Q)(N - C)}{N(N - C - Q)} \right)^{E+\frac{1}{2}} \left( \frac{E}{N - Q} \right)^Q \left( \frac{E}{N - C} \right)^C e^\delta \]

\[ = \left( \frac{(N - Q)(N - C)}{NE} \right)^{E+\frac{1}{2}} \left( \frac{N - C}{N} \right)^Q \left( \frac{N - Q}{N} \right)^C e^\delta \]

\[ = \left( 1 + \frac{QC}{NE} \right)^E \left( 1 - \frac{C}{N} \right)^Q \left( 1 - \frac{Q}{N} \right)^C e^\delta \sqrt{1 + \frac{QC}{NE}} \]

where

\[ e^\delta > e^{-\frac{1}{12(N-Q)+1}} e^{-\frac{1}{12(N-C)+1}} e^{-\frac{1}{12N}} e^{-\frac{1}{12E}} \tag{4-19} \]

\[ > e^{-\frac{1}{12E+1}} \tag{4-20} \]

\[ > e^{-\frac{1}{12E}} \tag{4-21} \]
since $12E < 12(E + C) + 1$ and $e^{-\frac{12E}{12E + C} + \frac{1}{12E}} > 1$. We know that the following inequality with some positive real number $\sigma$,

$$(1 + x)^\sigma > 1 + \sigma x$$

So,

$$
\left(1 + \frac{QC}{NE}\right)^E > 1 + \frac{QC}{N}
$$

Thus, the lower bound is obtained:

$$P_{\text{miss}} > \left(1 + \frac{QC}{N}\right)^Q \left(1 - \frac{C}{N}\right)^Q \left(1 - \frac{Q}{N}\right)^C \sqrt{1 + \frac{QC}{NE} e^{-\frac{1}{12E}}}$$

So, we are able to have success probability solely as a function of the constant replication factor $\alpha$, and independent of the number of the nodes.

4.3.2 Search Time

Search time can be analyzed by computing the average depth of the tree formed by the bounded broadcast presented in Section 4.2.3. When I assume that each node has a connection to its nearest neighbors on both rings as well as one shortcut connection to a node at distance $d$ with probability proportional to $1/d$, I can show that the expected depth of the tree is $O(\log^2 N)$. Our proof is analogous to computing the average time to reach a node using greedy routing as in [5, 47], while search time of Deetoo measures the average maximum depth of the local trees.
From the Kleinberg’s model, each node has a long-range contact at distance $r$ with probability $p_r = \frac{1}{\log N} \frac{1}{r}$ as in Figure 4-13. Let $p_m$ be the probability that node $a$ has a long-range contact in the middle, mL region (node $r$). Then, $p_m$ is inverse proportional to the distance between $a$ and $r$.

\[
p_m = \sum_{r=\frac{1+mL}{2}}^{\frac{1+mL}{2}} p_r
\]

\[
= \sum_{r=\frac{1-mL}{2}}^{\frac{1+mL}{2}} \frac{1}{\log N} \frac{1}{r}
\]

\[
\approx \frac{1}{\log N} \log \left( \frac{1 + m}{1 - m} \right)
\]

Since mL is a subset of L, $m \in (0, 1)$. Let $T(i)$ be the search time in region of size $i$.

Each long range connection either goes to a node in the middle region or not. If I make it into the middle region, the remaining area to search is at most the length of $\frac{1}{2}L + \frac{m}{2}L$.

Thus, if the long-range connection goes to the middle, the time is $T(L) \leq 1 + T\left(\frac{1+m}{2}L\right)$.

Otherwise, I go to the next neighbor so $T(L) = 1 + T(L - 1)$. I know that $T(L - 1) < T(L)$.

If $p_m$ is the probability of making it into the middle, on average, I need to try $1/p_m$ neighbors before I am likely to find a connection to the middle. Putting this together, the average search time is $T(L) \leq \frac{1}{p_m} + T\left(\frac{1+m}{2}L\right)$. The first part of the right side of the equation represents time to reach a connection in the middle (x in Figure 4-13) and the second part is time for the rest of the region (y in Figure 4-13). For the rest of the region, at most $\gamma$ more steps are required to cover the whole region L. I know $T(\log N) \leq \log N$.

Solving for $\gamma$ such that $\left(\frac{1+m}{2}\right)\gamma L = \log N$,

\[
\gamma \leq \frac{1}{2} \log \left( \frac{\alpha N}{\log \left( \frac{1+m}{2} \right)} \right)
\]
The maximum depth is decided to $\gamma$ steps multiplied by the number of the nodes to reach a connection in the middle.

$$T(L) \leq \frac{1}{p_m} \gamma \leq \frac{\log N}{\log \left( \frac{1+m}{1-m} \right) \log \left( \frac{1+m}{2} \right)} \leq \frac{\frac{1}{2} \log^2 (\alpha N) - \log \alpha \log (\alpha N)}{\log \left( \frac{1+m}{1-m} \right) \log \left( \frac{1+m}{2} \right)} \tag{4-30}$$

I calculate an upper bound with the above inequality by getting the maximum of the denominator. At a value of $m \approx 0.517$, the denominator of the right side of the above inequality has a minimum. In consequence, the upper bound for $T(L)$ is minimized.

I verify this claim using simulations which are presented in Figure 4-17. For the comparison, the minimum of upper bound, where $m$ is set to 0.5, is calculated and compared with the simulation result.

### 4.3.3 Stabilization Cost

I regard the number of objects that need to be copied to a newly joined node as the stabilization cost because there is no other major message transmission except this copying for a leaving or joining node. I assume that the algorithm checks whether an object is already replicated from either neighbor or not. If a node identifies the object, it discontinues the object transmission. This ensures that a node prevents unnecessary message generation and a network saves bandwidth. Objects which already exist in a new node as well as out of object’s range are excluded from transmission. A new node examines each object’s range whenever it is ready to be copied. In P2P networks, network size often varies. Thus, as the network size changes, the object’s range is recalculated. Caching in a redefined range constrains the number of cached replicas to remain $O(\sqrt{N})$.

Let $C_s$ represent the stabilization cost, $k$ the number of unique objects in the network, $N_1$ the left neighbor, and $N_2$ the right neighbor. Let $\eta_i$ symbolize a probability of
the object in node $N_i$. $\eta_{ij}$ is a probability of the object in both $N_i$ and $N_j$ and $\eta_{ij|i}$ denotes a probability of the object in $N_j$ given that the object is in $N_i$.

\[
C_s = kP(\text{object in } N_1 \text{ or } N_2) \quad (4-33)
\]
\[
= k(P(\text{object in } N_1) + P(\text{object in } N_2) - P(\text{object in both } N_1 \text{ and } N_2)) \quad (4-34)
\]
\[
= k(\eta_1 + \eta_2 - \eta_{12}) \quad (4-35)
\]
\[
= k(\eta_1 + \eta_2 - \eta_1\eta_{2|1}) \quad (4-36)
\]

The probabilities, $\eta_i$ and $\eta_{ij}$, are calculated as follows:

\[
\eta_1 = \eta_2 = \frac{L}{B} \quad (4-38)
\]
\[
= \frac{\sqrt{\alpha N d_{\text{ave}}}}{d_{\text{ave}} N} \quad (4-39)
\]
\[
= \sqrt{\frac{\alpha}{N}} \quad (4-40)
\]
\[
\eta_{12} = \eta_1\eta_{2|1} \quad (4-41)
\]
\[
= \sqrt{\frac{\alpha}{N}} \left(1 - \frac{d_{\text{ave}}}{L}\right) \quad (4-42)
\]
\[
= \sqrt{\frac{\alpha}{N}} \left(1 - \frac{1}{\sqrt{\alpha N}}\right) \quad (4-43)
\]

Thus,

\[
C_s = k \left(2 \sqrt{\frac{\alpha}{N}} - \sqrt{\frac{\alpha}{N}} \left(1 - \frac{1}{\sqrt{\alpha N}}\right)\right) \quad (4-45)
\]
\[
= k \left(\sqrt{\frac{\alpha}{N}} + \frac{1}{N}\right) \quad (4-46)
\]
\[
= O \left(k \sqrt{\frac{\alpha}{N}}\right) \quad (4-47)
\]

where $d_{\text{ave}}$ denotes the average distance between two nodes, $N$, $B$, and $L$ represent the number of nodes, the ring distance (the average distance times total number of nodes),
and the length of bounded broadcasting range (the average distance times the number of nodes in the range), respectively.

### 4.4 Simulation Results

In this section, I confirm the theoretical results discussed in the previous section via simulations. I used Netmodeler [45] for simulating the Deetoo algorithm. In the topology model on top of Netmodeler, each bin may be occupied by at most one node and the nodes form two rings, one for the cache and the other for query. Then, each node is connected to a long-range neighbor according to the inverse $r^{th}$-power distribution for the small-world network model. For simplicity, the size of the query range is assumed to be the same as the size of the cache range. All caching and querying processes are initiated on randomly selected nodes and messages are broadcasted within randomly chosen ranges.

#### 4.4.1 Performance Evaluation

In the simulations the size of the address space is set to 32 bits which has $2^{32}$ bins. I count the number of hops as an indication of communication cost and assume that the per-hop communication costs for cache and query are the same. Our simulations are performed on networks of size 10 to $10^6$. The number of columns, $C$, is chosen such that $C$ is equal to the number of rows, $Q$, $C^2 = \alpha N$. Thus, I simulate various values of $\alpha$ to observe the effect of increasing or decreasing the cache or query range size ($\alpha = \frac{C^2}{N}$). I performed simulations with $\alpha=0.1$ to 5.0, with intermediate steps of size 0.1. 100 string objects were generated by using a random string generator. Each object was initially inserted on a uniformly randomly selected node. For queries, I count only exact matching objects even though Deetoo can perform partial matches. Each test was repeated 100 times and the results were averaged. Figure 4-14 shows that query success probability is constant regardless of the network size. The constant success rate is desirable since Deetoo can perform the search with preferred success probability by adjusting broadcasting range with different $\alpha$. The success probabilities scale as

56
1 − e−α. In the legend of the figure, “th” is theory, the other is simulated. In the figure, I also compare the simulation results with the theoretical results. I observe that the larger the α is, the higher the success probability is. In Figure 4-15, query success probability declines exponentially as expected \( \exp(-\alpha) \). Figure 4-16 shows communication cost with respect to network size. Query cost scales \( O(\sqrt{N}) \) as I analyzed. “th” is for the theoretical result, and the other is the simulation result in the figure. There is a trade-off between success probability and the communication cost. As I increase a replication factor, α, I achieve higher success probability in the expense of communication cost.

For the search time, I measured the number of out-of-range links before finding a node in the range (by greedy routing) and the depth of the multicasting tree. Message transmission times at each link are assumed to be all the same. Figure 4-17 compares simulation results with calculations with α=1. As mentioned in Section 4.3.2, 0.5 is substituted for m. Note that x-axis scales logarithmically. Our simulation shows a loose upper bound for search time, but it is obvious that the scaling of simulation result is more than \( \log N \), but less than \( \log^2 N \).

4.4.2 Numerical Analysis for Success Probability

As described in Section 4.3.1, the success probability remains constant regardless of the number of nodes in Deetoo network. However, the accuracy of network size estimation affects success probability. The simulation results show slight trend downward of success probability as network size grows. The main reason for the slight decrease of success probability is that network growth makes variance of size estimation increase. In a bigger network, there are higher chances for a node to underestimate its network size relative to a small network. Underestimation is much more harmful for success probability because underestimation of the network size results in a narrower caching or querying range for bounded broadcasting. With the estimated network size distribution, I analyzed the impact of estimation on success probability numerically. I define \( w_c \) and \( w_q \) as the width of the the columns for caching
and querying respectively. Since different nodes will initiate a cache and a matching query, I can consider what happens when the two nodes have difference estimates of the network size.

\[ w_c \sqrt{B} = \sqrt{\alpha N} \quad (4-48) \]

\[ w_c = \sqrt{\frac{\alpha N}{B}} \quad (4-49) \]

Similarly, \( w_q = \sqrt{\frac{\alpha N}{B}} \). Thus, the number of bins in overlap is:

\[ w_c w_q = \alpha \frac{\sqrt{N_1 N_2}}{B} \quad (4-50) \]

Assume that \( N_1 \) and \( N_2 \) are network size estimation for cache and query, respectively. The success probability given a network size \( N \) is:

\[ P_{\text{success}}|N = \left( 1 - \frac{N}{B} \right)^{\alpha \frac{\sqrt{N_1 N_2}}{B}} \quad (4-51) \]

\[ = \left( 1 - \frac{N}{B} \right)^{\frac{N}{B} \left( \alpha \frac{\sqrt{N_1 N_2}}{N} \right)} \quad (4-52) \]

\[ \approx 1 - e^{\alpha \frac{\sqrt{N_1 N_2}}{N}} \quad (4-53) \]

Finally, I calculate the success probability with given network size and estimated distribution for both cache and query. To do this, I need to know the probability of estimating the size of the network given the actual size, \( P(N_1|N) \):

\[ \langle P_{\text{success}}|N \rangle = \sum_{N_1=0}^{\infty} \sum_{N_2=0}^{\infty} P(N_1|N) P(N_2|N) \left( 1 - e^{\alpha \frac{\sqrt{N_1 N_2}}{N}} \right) \quad (4-54) \]

Using the simulation data to obtain \( P(N_1|N) \), Figure 4-18 compares success probabilities among theory, simulation, and numerical evaluation. The figure shows that both the numerically evaluated success probability and the simulation results for success probability decrease slightly as network size grows compared to the theoretical result, which assumes a global knowledge of the network size. Bigger error margin for
a bigger network in estimating network size explains slight loss of success probability.

Even though there is some noise in the simulation results, I see that the simulation result matches the numerical evaluation nearly exactly: this is the source of the error. Mean square error between numerical evaluation and simulation result is 3.93e-05.

4.4.3 Robustness

Due to the nature of P2P networks, each peer can leave or fail at any time. Therefore it is important to analyze the effect of node failures. I will show that data objects are still accessible without generating excessive managing overheads under dynamic networks. I extended Netmodelerto model massive churn on nodes. The simulator setting is described below. I set the network size first. Each node joins the network sequentially following a uniform distribution. Upon completing network formation with a given size, each node repeatedly leaves and joins. A node’s rejoin time and leave time is exponentially distributed with same mean. The consequence of this distribution makes the average number of alive nodes in the system remains half of total number of the nodes originally joined. As seen in the Figure 4-19, a node's state transits between ON and OFF with probability of 0.5 because join time and leave time distribution has same average. In other words, the probability that each node is active at time t is $\frac{1}{2}$. In the simulation setting, every node stays to be turned ON until network size reaches a given number. It takes time a network size is saturated at the half of initial network size on average.

After all nodes joined network, 100 data objects were inserted. Queries were executed 100 times per data object. Both the cache events and the query events occur in a time distributed exponentially. For simplicity, a replication factor, $\alpha$, is set to 1 for the following simulations under churn.

Figure 4-20 demonstrates how the churn affects success probability. Note that success probability remains constant. Also, the success probability still follows the
theory as described in Section 4.3.1. This result shows that Deetoo is robust against network dynamics.

Under the heavy churn in the network, message transfers for maintenance purposes are not negligible. When a new node joins the network, it is responsible for maintaining data objects if its address is within objects’ range. I counted what fraction of the total objects were transferred to a new node. Assume that there are \( k \) objects, and each are replicated over \( \sqrt{N} \) nodes, I have \( k \sqrt{N} \) total objects in the system. These objects are placed on nodes in the range selected uniformly at random, so the expected number of objects on each node is \( k/\sqrt{N} \). Thus, joining cost is directly proportional to \( 1/\sqrt{N} \).

Figure 4-21 shows that how many replication occurred per unique data object after joining new nodes. For the measure of stabilization cost, the size of network is set to 1,000 nodes initially. Note that stabilization cost grows not as time continues, but as the number of unique objects increases. Unlike DHT-based stabilization, the process is much simpler, and costs less.

With very low maintenance cost, Deetoo is capable of efficient data retrieval even under heavy churn.

4.5 Applicable Searches

Structured P2P systems provide an efficient key-value look-up due to their hash table data structure. Despite their efficiency, a DHT’s data structure makes it challenging to answer new types of queries. When a new type of object or query is introduced, network topologies or caching strategies need to be redefined on structured systems. By comparison, unstructured systems can control new objects or queries without any modification. Unstructured systems make it possible to insert and retrieve different kinds of objects simultaneously over the same network. The object type may be text, meta-data, image, audio, or video. For example, Deetoo could be used for regular expression matching, content-based search to find similar multimedia objects to resolve
some query object, or to perform SQL queries, XQuery or XPath searches of XML documents. In the following subsection I describe the search process using regular expressions on top of Deetoo.

In this section, I present a practical search example which is one of many applicable searches with Deetoo. Regular expressions are used for searching strings of the text based on patterns. Objects might be strings, meta data of multimedia which are composed of strings, or objects’ titles. Managing data is not easy in decentralized and distributed networks which have no central control. In a distributed manner, there are critical issues to be overcome. One issue is how to avoid transmission of multiple identical data from different peers to save bandwidth usage. Another issue is how to delete all the duplicated objects from the entire network when they are obsolete. Because objects are spread over the network due to Deetoo’s caching property through bounded broadcasting, it is very important that all the replicated objects should be deleted when an original object needs to be deleted. It is assumed that each node maintains object records which include each object’s ID, range information (start address, end address), and a replication factor ($\alpha$) as well as content. The ID identifies data object and it is useful for deleting objects. The range information indicates the first address and the last of bounded broadcasting in a caching network. I explain how to apply operations for regular expression search on top of Deetoo.

4.5.1 Object Insertion:

Objects or files are easily inserted into a network by bounded broadcasting as described in Section 4.2.3. $O(\sqrt{N})$ replicated objects reside in the network after insertion. Once an object is cached in the network, Deetoo requires to adjust the object’s range information as the network size changes. Every time a new node joins the network, the node estimates network size. Before it copies an object from its neighbor, it calculates the object’s range based on the estimated network size. With a replication factor ($\alpha$) in the object’s record and the estimated network size ($N_{\text{est}}$), one can calculate
a new range \( R = \sqrt{\alpha \frac{B}{N_{\text{est}}}} \). The midpoint of the range \( M \) can be obtained by adding start and end addresses of the range from the object’s record and dividing by 2. Finally, \( M + \frac{R}{2} \) and \( M - \frac{R}{2} \) are the new start and end address of the range. Then it compares it to the object’s range information in the record. In accordance with a recalculated range, copied objects may be discarded from nodes or added to nodes.

### 4.5.2 Search

When a node tries to find objects in corresponding to a given pattern, the node bounded-broadcasts the pattern in a randomly selected range. Users are able to select a searching method. There are three searching options, simultaneous, sequential, and deleting.

Simultaneous search uses bounded broadcasting. A pattern is transmitted to the nodes in the range. All matching results are transferred to a querying node simultaneously. This search method runs in time \( O(\log^2 N) \) and the querying node receives multiple replies.

Sequential search is designed to avoid multiple answers from multiple peers. A query sequentially visits a node in a tree under simultaneous option. First, a query starts bonded broadcasting to form a local tree. Children nodes having the matching object send results back immediately and the parent node waits for the first response from its children nodes. Upon receiving the first response, the parent node stops waiting for and transfers the result to a querying node. This process takes place recursively like depth-first search (DFS). Figure 4-22 is an example of a local tree formed by bounded broadcasting. Node 0 sends a query to node 1 instead of sending to all children node(node 1,3,8,13). If no response from node 1 is received, it sends a query to node 3. Node 0 obtains the first response, node 0 stops action and forwards the response to a querying node immediately.

The sequential search option leads to an increase the searching time because nodes in the tree are visited sequentially. The search time for sequential search is linear
in the range size, which is $O(\sqrt{N})$ in network size, while the simultaneous search takes at most $\log^2 N$ as described Section 4.3.2. At the expense of increased searching time, Deetoo prevents the generation of a large number of messages, and thereby it saves bandwidth utilization. This option is especially useful for a popular object or a loosely defined regular expression search.

The deleting option is used to delete objects from a network. The usage of deleting option is described in the following paragraph.

4.5.3 Object Deletion:

Data deletion is more complicated than aforementioned operations. This is because data are distributed over the network. In addition, peers cannot exactly identify how many replicated data exist in the network. Thus, the Deetoo algorithm requires two stages of operations for deleting data objects from a network. First, Deetoo searches with the deleting option in an arbitrary bounded broadcasting region in a querying ring. The deleting option distinguishes a search for deletion from a plain search. If there exists any node with a matching object, the node triggers the second stage, returning object’s ID back to the asking node. In the second stage, the node starts bounded broadcasting the DELETE command with the object’s ID using the node’s range information, and every node receiving the message discards the object whose ID matches. Unlike the first stage, the second stage takes place in a caching ring because range information indicates the caching addresses. I allow an asking node to use exact matching search rather than regular expression search. If regular expression search is utilized for the object deletion, certain similar objects that are not intended to be deleted can be eliminated. Consequently it would result in unexpected data loss.

4.6 Conclusion

In this chapter, I introduced Deetoo, a scalable routing peer-to-peer protocol for unstructured networks which provides efficient caching and lookup functionality with $O(1)$ query hit-rate, $O(\sqrt{N})$ replication, $O(\sqrt{N})$ query cost, and $O(log^2 N)$ search time.
Deetoo, allows objects to be updated and deleted, and can be used to handle general queries. Specifically, any problem that can be mapped onto selecting (with high probability) objects which match some query can be run over Deetoo.

![Figure 4-1. The caching and querying space.](image)

![Figure 4-2. Virtual ring 1 for caching.](image)

![Figure 4-3. Virtual ring 2 for querying.](image)

![Figure 4-4. The complete searching space.](image)
Figure 4-5. Tree generation by bounded broadcasting

Figure 4-6. Object injection

Figure 4-7. Bounded broadcast in the range

Figure 4-8. Completion of object replication

Figure 4-9. Initiation of a query for object o

Figure 4-10. Bounded broadcasting within Q

Figure 4-11. Object retrieval n(o).
Figure 4-12. Estimated network size distribution

Figure 4-13. Probability of long-range connection

Figure 4-14. Query success probability
Figure 4-15. Query miss rate

Figure 4-16. Communication cost for query
Figure 4-17. Search time

Figure 4-18. Success rate comparison
Figure 4-19. State transition diagram

Figure 4-20. Success probability under churn
Figure 4-21. Stabilization cost

Figure 4-22. Sequential search
CHAPTER 5
UNSTRUCTURED SEARCH ON PLANET-LAB WITH DEETOO

5.1 Introduction

P2P applications are highly scalable currently and dominate Internet traffic. One of the most popular applications in P2P is file sharing. An efficient search system is a key component in building many P2P applications. There are currently two types of P2P search techniques: structured and unstructured. Unstructured search systems using naive flooding or random walk are inefficient with respect to search cost since potentially every node in the network may have to be visited when searching for rare objects. On the contrary, DHT-based structured search systems take only logarithmic hops in the size of the network while guaranteeing object retrieval in the network. The disadvantages of using DHT-based approaches are higher maintenance cost and complexity. In addition to the higher maintenance complexity [2, 3, 48], DHTs do not have a good solution for general queries because it is hard to map interesting queries onto the DHT. However, building such a search system is not easy because, in most cases, to deploy a P2P-based system developers must start almost from scratch. In other words, developers must build an entire system including functions for sending Internet protocol packets, managing connections, and traversing NATs. For instance, to build Chord [3], 35,030 lines of code are required to build an entire system including a search function. Another P2P system, Pastry [48], needs 25,915 lines of code and Brunet [49] is composed of 34,369 lines. A search function is included in all currently deployed P2P systems, but the function is exclusively applicable to its own system. I built Deetoo with less than 1,000 lines of code on Brunet. Deetoo will enable system designers to plug in a query matching model and use an off the shelf P2P system. By adding less than 1,000 lines to an existing P2P system, developers can build a novel search system.
In Chapter 4, Deetoo [50] is proposed for general search system to address the problems in both unstructured and structured search systems. Deetoo is an unstructured search system built on top of structured P2P networks. The users can achieve constant query success probability without regard of network size. Users need only select a replication factor to satisfy their desired hit-rate/cost trade-off. Deetoo is more efficient than flooding-based query systems and supports various types of queries, such as regular expression queries, unlike DHT-based query systems.

In this chapter, I describe an implementation of Deetoo in a real P2P environment and compare to theoretical expectation as well as simulation results to verify the performance of Deetoo. I built Deetoo as service on top of Brunet [51] and deployed Deetoo-enabled nodes in Planet-Lab [52].

Planet-Lab is a global network research testbed that consists of 1089 nodes at 505 sites globally. Many researchers in academic institutions as well as in industrial research labs benefit from Planet-Lab to test their new technologies in several research fields, including P2P networks, distributed file systems, network performance measurement, network security, and many other areas. Developers can encounter many practical problems when they implement their proposed designs in a real network environment. Planet-Lab helps to address those kinds of problems by providing a real live testbed to help solve practical issues with the design. The experiments in this chapter are conducted on Planet-Lab using Brunet P2P overlays.

Brunet is a library for P2P networking, which provides the core services of routing, object storage/lookup, and overlay connection management supporting multiple transports (TCP, UDP, and tunnels) and NAT traversal. Many different types of services such as IPOP [53], WOW [54], Grid-appliance [55], and SocialVPNs [56] are already built on Brunet, and some of them provide their service as open software. While Deetoo can be built on top of many P2P applications, Brunet is used for the underlying substrate for the P2P network management in Deetoo implementation. Brunet’s network topology
follows Symphony [5] based on a one-dimensional ring similar to Chord [3]. Deetoo is implemented on top of Brunet as a service that provides object caching and querying functionality.

Deetoo’s computational component for caching and querying is implemented using MapReduce [57] approach. To handle large-scale, data-intensive computation in distributed systems effectively, parallel processing in data partitioning and computation is a good way to boost the speed of data processing. The MapReduce concept comes right from supporting such parallel computation. MapReduce is composed of two computation functions, Map and Reduce. Map transforms an input to a new value as an output of the same type. Reduce takes the results both from Map and Reduce as arguments and passes the resulting values with same key to the node’s ancestor. With MapReduce, a user needs to define map and reduce functions and the MapReduce platform handles the parallel execution and interaction among MapReduce modules. For the purpose of taking advantage of parallel processing, a MapReduce technique is used in the implementation of Deetoo on top of Brunet. The details of MapReduce in Deetoo are described in Section 5.2.3.

To measure the search performance with Deetoo, several metrics are used,

1. **Query success probability**: the number of successfully retrieved data over the number of search trials.
2. **Communication cost**: the number of nodes accessed by a caching node and a querying node for data replication and search, respectively.
3. **Search time (latency)**: the tree depth built by bounded broadcast in a given range
4. **Trade-off between cost and success probability**: caching cost versus query success probability
5. **Estimated network size**
6. **Load balancing**
5.2 System Model

While Deetoo is applicable to many ring-based P2P overlays, Deetoo’s functionality is built on top of Brunet P2P overlay networks. The use of Brunet overlay makes it easy to deal with many difficult issues related to network management including the management of connections and NAT traversals. The network topology of Brunet and how Brunet manages connections and routing are described in the section.

5.2.1 Brunet-Network Topology

The Brunet P2P library provides mechanisms for building and maintaining structured P2P networks of overlay nodes. The structure of Brunet is the same as Symphony’s structure, in which each node maintains connections with its nearest-neighbor nodes in P2P address space. These connections are called structured near connections. In addition to the structured near connections, each node also connects to \( k \) distant nodes. These nodes are called structured shortcut connections, and nodes at distance \( d \) with proportional to \( \frac{1}{d} \) are selected as shortcut connections. The shortcut connections contribute to the decreased access time for bounded broadcasting. With the help of shortcuts, the hop distance in a given range of bounded broadcasting is limited to \( O\left(\frac{1}{k} \log^2 n\right) \), given a network of \( n \) nodes \([47]\). Typically \( k \approx \log n \), so time is \( O(\log n) \).

The Deetoo service creates two virtual nodes on one physical overlay node. Each virtual node joins either a caching network or a querying network, and they share a list of cached objects. By receiving a caching message, a caching node writes an object’s information in the cache list. Caching nodes also conduct stabilization as connections change and as network size changes. The variation of network size estimation results in the change of the broadcasting range of cached objects. Thus, stabilization is required to maintain the correct number of replicas of each object. The querying node is responsible for responding to query messages. Upon receiving a query message,
the node first reads the query type and then executes a proper query resolving action. Figure 5-1 shows how two virtual networks are built on top of Brunet.

### 5.2.2 Network Size Estimation

The accuracy of network size estimation is important in distributed systems such as P2P networks. In Deetoo’s bounded broadcasting, the performance of Deetoo’s search algorithm depends largely on the accuracy of estimation because the range of bounded-broadcasting is determined by the estimated network size. To address the issue of network size estimation, a sequential update is introduced. First, a node calculates node density using only direct neighbors (one left and one right connection) after it joins a network. Based on the node density, the node gets the first estimation, which is \( N_0 \). In the next estimation step, the node sends a simple ping message to a \( \log N_0 \)-hop away node and retrieves the remote node’s address information; this gives \( N_1 \). Finally, it requests the estimation to its shortcut neighbors then takes a median. For both the caching and querying purposes, the ranges for bounded-broadcasting are determined by the final median estimation. The accuracy of the sequential estimation is compared with the actual number of nodes in the network in Section 5.4.

### 5.2.3 MapReduce-Based Cache/Query System

Map and reduce functions are popular paradigms in functional programming languages (e.g., Ruby, Python, and LISP). A map function transforms each element \( x_i \) in a collection \( (x_1, x_2, \ldots, x_n) \) into a different key element \( y_i \), thus creating a collection of intermediate elements \( (y_1, y_2, \ldots, y_n) \). A reduce function computes an aggregation \( z \) over this collection. Based on the concept of the map and reduce functional programming model, software frameworks (e.g., Hadoop [58] and Google [57]) have been developed to efficiently parallelize computations on large datasets. The map function usually works on (key/value) pairs to create intermediate (key/value) results. Reduce functions work on intermediate (key/value) results while aggregating intermediate values associated with the same intermediate keys. The framework distributes map and reduce tasks among...
nodes in a cluster to enable parallel processing, while users only have to specify the appropriate map and reduce functions associated with their computation without having to worry about details of distributed parallel job execution.

The implementation of Deetoo on top of Brunet consists of five functional modules. They include a P2P network module, a MapReduce core, a tree generation module, a map module and a reduce module. Figure 5-2 shows the architecture of MapReduce function modules. When a task (caching/querying) is assigned through the underlying P2P network's Remote Procedure Call (RPC), the MapReduce core module first generates a tree using bounded broadcasting. After a tree is successfully generated, map functions are executed in parallel at all the nodes in the tree. The MapReduce core in an initiating node starts the map-reduce process. While Map is processing its own function at a node, the node waits for the response from its child nodes. Upon receiving results from child nodes, the node reduces the job, then returns the reduced result to its parent node. When reduce is completed, the whole result is passed to the root node. Note that the root node in a tree is the initiating node for cache or query. Figure 5-3 illustrates a simple example of a MapReduce tree. $x_i$ is an argument for a map function. $z_j$ and the result of map are arguments for a reduce function. Let a map function be $f(x_i)$, a result of reduce be $z_i$, and a reduce function be $g(z_j, ..., z_k, x_i)$. The root node, $x_1$, executes its own map function and gets a result, $f(x_1)$. The child nodes $x_2$ and $x_3$ pass their reduce results. The map and reduce functions are executed simultaneously in the nodes at the same level. Finally, $x_1$ receives reduce results from $x_2$ and $x_3$ and it reduces with them and its own map result. The reduce results in leaf nodes $(x_4, x_5, x_6)$, which are the same as their map results in this example because they do not have any child nodes that pass any results.

The details of each module's functions is described in the following sections.
5.2.3.1 P2P network module

The P2P network module is responsible for handling new node joins and departures, connection management with neighboring nodes, and message routing. The current implementation of Deetoo is developed on top of Brunet. While the underlying network module for the Deetoo implementation is Brunet, Deetoo can be built on many other P2P platforms, such as Chord [3], CAN [2], and Pastry [48]. This can be done by building a 2-dimensional space, then cutting the space to intersect at one point for cache and query.

5.2.3.2 MapReduce core module

The MapReduce core module is responsible for distributing map and reduce functions using bounded broadcast. When a user initiates a MapReduce task, the request is conveyed to the MapReduce core module through the underlying P2P network’s Remote Procedure Call (RPC) module. The MapReduce core module checks the map arguments, the reduce arguments, and the broadcast region arguments. The map and reduce arguments will be passed to the map and reduce functions, respectively. Map arguments include an object to be inserted into a network in case of caching and an object and a query type to be passed as map arguments for querying. The results from child nodes and the result of map are passed as reduce arguments. The broadcast argument describes a bounded-broadcast region that the node is responsible for. The MapReduce core module disseminates the task to nodes which reside under its responsible region after manipulating the broadcast region argument appropriately.

5.2.3.3 Generating a tree module

The generating tree functional module makes use of bounded broadcast to replicate objects as well as to propagate query messages. With the existence of shortcut connections in a Brunet P2P network, the bounded broadcast builds a tree whose depth is short (≈ log N) compared to a naive ring-based P2P overlay. The detailed
operation of bounded broadcast that builds a multicast tree is described in [50]. The range of bounded broadcasting is selected at random, while the size of the range depends on the estimated network size and the replication factor, $\alpha$. The replication factor is a user-defined parameter to the generating tree functional module. By adjusting the replication factor, users are able to control the number of replicas in a caching network or manage query success probability in a querying network.

5.2.3.4 Mapping module

There are two separate map functions for caching and querying for the test. For caching, the map function inserts a data object into the node’s data list and returns true if data insertion was successful. In the querying purpose, map function searches a specified string from the data list. The map function for the query also supports regular expression search using the existing .Net library. The result of the map function is either success or failure if it is called for the caching purpose, while the map function for the querying returns the matching object. The map result is transferred to the node’s reduce function module as an argument. For the caching and the querying purposes, map functions are described as follows:

**Cache Map Function:** The pseudo-code for a cache map function is shown in Algorithm 1. The function takes an object to be inserted as argument. The node which runs the map function accesses its list of objects and checkes if the inserting object already exists in the list or not. The result of cache map function returns true or false depending on success or failure of insertion.

**Query Map Function:** Algorithm 2 describes the query map function. The function requires an object to be searched as an argument and it returns a list of matching objects. If the function cannot find any match, the query map function returns an empty list.
Algorithm 1 CacheMap

Require: object o // an object to be inserted
Require: Node n
1: object_list ← n.getObjects()
2: if o is not in object_list then
3:     object_list.put(o)
4:     return true
5: else
6:     return false
7: end if

Algorithm 2 QueryMap

Require: object o // an object to be searched
Require: Node n
1: object_list ← n.getObjects()
2: result = []
3: for all v ∈ object_list do
4:     if o matches v then
5:         result.Add(v)
6: end if
7: end for
8: return result

5.2.3.5 Reducing module

In the reduce functional module, the map results from the working node and the reduce results from the child nodes are passed to the reduce module as inputs. Assume that a node has \( n \) child nodes in a tree. The working node sends a message to its child nodes in the tree that is built using bounded broadcast. Thus, the node expects \( n \) results from its child nodes and one result from its map module. By reducing \( n \) reduce results and one map result, the node transfers \( n + 1 \) reduce results to its parent node in the tree. In the caching process, the boolean results which indicate success or failure of caching, the number of nodes and the time spent in terms of tree depth are passed as arguments to the reduce function. In the querying process, the boolean results are replaced by the matching results. The reduce function for regular expression match concatenates all the reduce results and passes the results up the tree. By the end of each process over the network, a user collects information about the total number of nodes visited and tree
depth in addition to caching or query result. As reduce is processed at each node in a tree, the results are propagated back through the tree. For the caching and the querying purposes, reduce functions are described as follows:

**Cache Reduce Function:** Algorithm 3 explains how a cache reduce works. When a cache reduce is called by a node, it collects all the results from the node’s children in a tree and they are transferred to the cache reduce function as arguments. In the test setting, the child results are a list of boolean values that indicate the success or failure. In addition, the result of map from the node is passed as a reduce argument. If one of the arguments is NULL, the reduce function returns only a valid result from either map or child result. The function combines the map result with the results from child nodes, then return the combined result.

```
Algorithm 3 CacheReduce

Require: Map Result m, // [true, false,...,false]
Require: Child Result c // [false,true,...,true]
1: result_list ← []
2: if m is NULL then
3:     result ← c
4: else if c is NULL then
5:     result ← m
6: else
7:     result ← m.combine(c)
8: end if
9: return result_list
```

**Query Reduce function:** The pseudo-code for the query reduce function is shown in Algorithm 4. This is almost the same as the cache reduce function. The difference is that the result from both child node and map is a list matching object.

### 5.3 Experimental Environment

In this section, the performance of Deetoo is confirmed via experiments in Planet-Lab. Deetoo nodes, each composed of two virtual Brunet nodes, were deployed. The virtual nodes form two different virtual overlay networks, one for caching, the other for querying. Brunet framework manages connections among nodes and provides the
Algorithm 4 QueryReduce

Require: Map Result \( m \), // a list of matching objects

Require: Child Result \( c \) // a list of matching objects

1: \[ \text{result}\_\text{list} \leftarrow [] \]
2: \[ \text{if} \ m \text{ is NULL then} \]
3:  \[ \text{result} \leftarrow c \]
4: \[ \text{else if} \ c \text{ is NULL then} \]
5:  \[ \text{result} \leftarrow m \]
6: \[ \text{else} \]
7:  \[ \text{result} \leftarrow m.\text{combine}(c) \]
8: \[ \text{end if} \]
9: \[ \text{return} \ \text{result}\_\text{list} \]

mechanism for UDP packet transport. Also, it supports NAT and firewall traversals.

The Deetoo search system was implemented on more than 400 Planet-Lab nodes. For the experiment, 100 unique string objects were cached over the network. Then, the querying process was repeated 100 times for each unique object.

Test was performed with two different values of replication factor \( (\alpha) \) to observe the effect of increasing or decreasing the caching or querying range size. The replication factor in the experiments was set to 1.0 and 3.0. Note that the replication factor is the only parameter to affect a hit rate, which is \( e^{-\alpha} \). \(< \alpha > \) nodes get both cache and query.

The matching function used regular expression match for the entire test. All caching and querying processes were initiated on randomly selected nodes and messages were broadcasted within randomly chosen ranges.

5.4 Evaluation and Results

The evaluation is focused on (a) accuracy of network size estimation, (b) query success probability, (c) latency in terms of the depth of a tree generated by bounded broadcast, and (d) communication cost.

5.4.1 Network Size Estimation

In the experiment, the moving average of sample size 100 was used for evaluation. In other words, the past 1000 results were averaged at the time indicated. The moving average is good for showing the variation of a small subset of results given a very
large number of samples. Figures 5-4 and 5-5 show how accurately the network size estimator performed. The results of the network size estimation were slightly larger than the actual network size. However, the deviation was very small. The difference between measured network size in each node and the actual network size resulted from the non-uniform distribution of nodes’ addresses.

5.4.2 Query Success Probability

The query success probability was measured. Figure 5-6 shows the query success probability as a function of measured time with a replication factor of 1.0 and Figure 5-7 shows the probability when the replication factor was set to 3.0. The variation is observed in each experiment. The query success probability is only dependent on the replication factor and the network size. Considering a deterministic replication factor where a user can select at his desired performance, the network size is the only influential factor for the query success probability. Each result was compared to the theoretical result that is approximately $1 - e^{-\alpha}$.

As seen in Figures 5-6 and 5-7, the query success probability is almost constant in time. The noise in the figures results from the variance of network size estimation since the estimated network size at the time of cache or query decides the range of bounded broadcast. The results were comparable to the theoretical results. The constant success probability is desired since Deetoo can perform the search with preferred success probability by adjusting broadcast range with different replication factors. As expected, the larger the replication factor, the higher the query success probability.

5.4.3 Latency

The latency in the test is measured by counting tree depth. In the calculation of latency, the number of out-of-range links before finding a node in the range by greedy routing is added to the tree depth. According to [47], the latency of bounded broadcast in a given network size, $n$, is bounded by $O(\frac{1}{k} \log^2 n)$, where $k$ is the number of shortcut links. In Brunet, each node maintains approximately $\log n$ shortcut connections.
Thus, the tree depth is $O(\log n)$. Figure 5-8 and 5-9 compare theoretical results with experimental results when $\alpha$ is 1.0 and 3.0, respectively. The experimental results show shorter latency because the number of shortcut connection is greater in Brunet than in calculation. Considering that MapReduce is executed in parallel in a tree, a shorter latency produces a more desirable responsiveness. The actual latency ($L$) for the querying is measured. The latency between a pair of nodes is not measured but assuming that the per-hop latency ($L_{\text{hop}}$) between two nodes is constant, per-hop latency can be calculated from the total latency and the depth of tree ($l_D$).

$$L_{\text{hop}} = \frac{L}{l_D}$$

Table 5-1 shows the measured and calculated results of latency, communication cost, tree depth, and per-hop latency. For $\alpha = 1.0$, the latency will be increased to 9.42 (sec) (per-hop latency $\times$ communication cost) if a query message is delivered sequentially instead of using bounded broadcast. Sequential message relay takes 4 times more time than bounded broadcast. Figure 5-10 shows the measured latency of queries.

5.4.4 Communication Cost

Communication cost is evaluated as the number of nodes reached by a caching or a querying message. Figures 5-11 and 5-12 shows the communication cost with different replication factors. Only 5% of nodes needed to be accessed by the :w searching process to achieve over 60% of query success probability and less than 9% of nodes were visited to satisfy over 90% of query success probability. Since the size of UDP packet for a string object caching is 384 byte and the average number of messages for $\alpha = 1.0$ is 20.54, approximately 8 KB of total data is transferred for a single cache. Figure 5-13 illustrates the total bytes transferred in the network for each query.

5.4.5 Replication vs. Success Probability

There exists a trade-off between the number of replicas per object and query success probability. The number of replicas varies from one object to another even
if the given replication factor is the same for each object. The number of replicas is
determined based on the network size estimation. Some nodes may over estimate and
others may under estimate the size of the network. If an object is inserted from a node
whose estimation is less than the actual network size, the object is replicated more.

The test counted the number of replicas per each unique objects after stabilization.
For the query success probability, the number of successfully resolved queries per each
object was counted and averaged. Figures 5-14 and 5-15 show the trade-off between
replication and query success probability with $\alpha$ of 1.0 and 3.0, respectively. I observed
that query success probability increased as the number of replicas increased until query
success probability reached 100%.

5.4.6 Load Balancing

Let $m$ be the number of objects cached, $N$ be the number of nodes, and $C$ be the
communication cost. It is known that $C = \sqrt{\alpha N}$ from the equations (4–4) and (4–5) in
Chapter 4. Thus, the number of objects per node, $m_{ave}$, is:

$$m_{ave} = \frac{m \times C}{N}$$

$$= m \sqrt{\frac{\alpha}{N}}$$

I measured the number of objects per node from 368 Planet-Lab nodes ($N$). 500
objects ($m$) were cached over the network with a replication factor 1. As seen in Figures
5-16 and 5-17, the load is evenly distributed. The average number of objects in a node
is 24.5 and the standard deviation is 5.46. The theoretical expectation for the number of
replicated objects per node is 26.06 if the replication factor is set to 1. The reason for the
mismatch between the measured result and the theoretical result came from the over
estimation of the network size. The average of estimated size was 395.6, but the actual
size was 384. The over-estimation resulted in the smaller number of replicated objects.
5.5 Conclusion

In this chapter, the Deetoo search technique is evaluated and tested on Planet-Lab to verify the performance of Deetoo in a real P2P network environment. The Deetoo search module is developed on the existing P2P overlay. The test of the regular expression search with the Deetoo confirmed the general type of query resolving ability.

The test results show that Deetoo’s caching and querying performance is very close to both theoretical and simulation results. The performance of Deetoo on real networks shows that Deetoo is an attractive solution for unstructured searches on top of structured P2P networks. The developed Deetoo search component on top of P2P substrate in this work is applicable to other existing P2P overlay networks.
Table 5-1. Latency

<table>
<thead>
<tr>
<th>metric</th>
<th>$\alpha = 1.0$</th>
<th>$\alpha = 3.0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>latency (sec)</td>
<td>2.29</td>
<td>2.71</td>
</tr>
<tr>
<td>depth of tree</td>
<td>5.0</td>
<td>6.26</td>
</tr>
<tr>
<td>per-hop latency (sec)</td>
<td>0.46</td>
<td>0.43</td>
</tr>
<tr>
<td>no. messages</td>
<td>20.48</td>
<td>34.38</td>
</tr>
</tbody>
</table>

Figure 5-1. P2P (Brunet) network overview
Figure 5-2. MapReduce function modules for query system

Figure 5-3. Simple example of map-reduce tree
Figure 5-4. Estimated network size ($\alpha = 1.0$)

Figure 5-5. Estimated network size ($\alpha = 3.0$)
Figure 5-6. Query success probability ($\alpha = 1.0$)

Figure 5-7. Query success probability ($\alpha = 3.0$)
Figure 5-8. Latency in terms of tree depth ($\alpha = 1.0$)

Figure 5-9. Latency in terms of tree depth ($\alpha = 3.0$)
Figure 5-10. Measured Latency

Figure 5-11. Communication cost ($\alpha = 1.0$)
Figure 5-12. Communication cost ($\alpha = 3.0$)

Figure 5-13. Total bytes transferred for each query
Figure 5-14. Communication cost for caching vs. query success probability ($\alpha = 1.0$)

Figure 5-15. Communication cost for caching vs. query success probability ($\alpha = 3.0$)
The number of objects per node

Figure 5-16. Load distribution

The number of objects per node

Figure 5-17. The number of cached objects per node
6.1 Introduction

The core of most P2P systems is an efficient search algorithm. In Chapter 4, Deetoo is proposed to provide an efficient, scalable, self-organizing, and optimal search algorithm in terms of trade-off between caching cost and querying cost. In addition, Deetoo can resolve any types of queries since the data objects are not structured. The address for a new node is selected at random. Deetoo chooses columns (or rows) for bounded broadcasting at random, even though users decide the number of columns (or rows) by providing a replication factor. Users can control a hit rate by increasing or decreasing a replication factor, but users also have to pay for more bandwidth. Deetoo can only achieve a 100% success rate for search by broadcasting a query message over the entire network, however, in that case, the network may be overloaded with query messages and the network is not scalable. Due to the randomness of Deetoo, it does not guarantee a hit. There is always a chance not to resolve a query. Deetoo is applicable to most P2P systems such as file-sharing or resource-discovery that are not strict with respect to the hit rate. Those applications often require multiple query results. Even if there was no match after a querying process, the applications simply send another query message and the message is broadcasted in another area. However, some applications require the highest level of hit rate, or even a 100% hit rate. One example of such an application is a P2P database. To address the problem of Deetoo’s inability to achieve 100% hits, Exact Deetoo is suggested in this chapter.

The Deetoo algorithm succeeds with a certain probability that meets a user’s needs. A user can achieve a constant success probability regardless of the size of the network by adjusting a replication factor. However, here I present a way to make a change so that the query process always reaches a successful result without paying extra communication cost (communication cost remains $O(\sqrt{N})$). One way to achieve 100%
success probability is to guarantee that every overlapped bin (a place where a cache and query column overlap) has at least one node in it. While Deetoo's join cost remains in very small, Exact Deetoo requires slightly higher joining cost. However, once nodes are joined in a network, communication cost both for caching and querying follows $\sqrt{N}$ and the query always succeeds.

A node join algorithm in Exact Deetoo is implemented differently than in randomized Deetoo, while the Exact Deetoo shares Deetoo's routing technique and network topology. The basic assumption of Deetoo is that each address bin is occupied or is empty. Each node randomly selects a new address in Deetoo. In Exact Deetoo, a new address is carefully selected by algorithm. The Exact Deetoo algorithm chooses a new address such that each box maintains the minimum number of nodes. Because each box is a cross-sectional area of caching range and querying range, a query always succeeds by maintaining at least one node in each box. In Exact Deetoo, a bin is expanded to contain multiple nodes. The bin is now called a ‘box’. A box is a collection of address bins in a rectangular space as seen in Figure 6-10. A box is the same concept as the cross-section of columns and rows in Deetoo (the area $A$ in Figure 4-1). The number of boxes and the width and height of the boxes vary according to the network size and the number of boxes are closely related to the cost of joins. The effect on cost is discussed in the following section. The maximum and minimum numbers of nodes in a box is set by the user. If the number of nodes in a box reaches the maximum allowed, the node tries to find another box that is not full in the same column or row. If all the boxes in the same row or column are filled with the maximum number of nodes allowed, the boxes in the row/column split into two sub-boxes. On the other hand, if one or more nodes leaves a box such that the box does not maintain the minimum number of nodes, a node in the box requests a node that resides in a box with many nodes to move into the box. Two adjacent columns/rows are merged when many boxes contain fewer nodes than the minimum. As the network grows, Exact Deetoo makes each box
smaller. Since Exact Deetoo assigns a new node’s address such that any box retains appropriate number of nodes, users’ queries always succeed. The main contributions of Exact Deetoo are: 1) By using Exact Deetoo, queries always succeed. 2) The costs for both publishing and searching remain \( O(\sqrt{N}) \). 3) Exact Deetoo is able to resolve any kind of query such as a regular expression search since the data objects or query messages are not structured.

In this chapter, Exact Deetoo algorithm is evaluated through simulation. The metrics for the evaluation are box properties (number of boxes in the network, number of average nodes in a box), cost for new node joins, caching and querying cost, search time in terms of tree depth, and load balance.

### 6.2 The Algorithm

First the design of Exact Deetoo is described. Then, in Section 6.2.1 the way which a new node joins a network is described. The basic form of Exact Deetoo is the same as Deetoo: a 2-dimensional matrix space. At any point in time, the space is dynamically split among all the nodes such that every node owns its individual zone within the overall space. The zone is shared by several nodes to prevent an empty zone after subsequent splitting as network size grows. The zone is called a ‘box’ for the rest of the chapter. The box is the same concept of cross-sectional region in Deetoo. For example, Figure 6-1 shows a 2-dimensional 4-by-4 space. The boundaries of each box are represented by column number and row number. The box A has a boundary of \([w/4, 2w/4]\) in the column and \([2w/4, 3w/4]\) in the row, assuming that the size of the address space is \(B = w^2\). The box A contains 3 nodes including the node \(n\) and A is associated with \(n\) until A is split or merged. The difference is that Exact Deetoo always maintains the minimum number of nodes in every box in order to be able to resolve every single query. As in Deetoo, each node has two addresses, one for the cache network and the other for the query network. The purpose of using two addresses is to create an ordering on the cache ring increasing along the columns and on the query ring increasing along
the rows. The address for a given node in the query ring must be the transpose of the address on the cache ring. Nodes in the same column have adjacent addresses in the cache network. On the query network, nodes in the same row have adjacent addresses. Adjacent nodes make a near connection to each other. In the figure, each box contains at least 2 nodes.

Nodes in the Exact Deetoo self-organize into an overlay network that represents this virtual space. A node maintains its near and shortcut neighbors in each ring. Also, a node keeps track of the box associated with that node so that the node is always able to identify the range of broadcasting. For instance, when \( n \) inserts an data object, the object will be cached at all the nodes between column \( w/4 \) and column \( w/2 \). The broadcasting range for caching is decided by the box \( A \), and the range is from \( w/4 \times w \) to \( (w/2 - 1) \times w + (w - 1) \). Similarly, a query message must be sent to all the nodes between row \( w/2 \) and row \( 3w/4 \). The range for a query broadcast is to be from \( w/2 \times w \) to \( (3w/4 - 1) \times w + (w - 1) \). Note that each node utilizes the caching address, in case of caching and the querying address, which is transposed form of caching address in case of querying.

### 6.2.1 Join

As described above, the entire address space is divided by boxes that each contains a certain amount of nodes. It is critical that a newly joining node finds a proper place to achieve 100% query success rate. It cannot be allowed that any box in the network is empty. This is done by selecting an address for the node in that the node fills a box or splits a box if the box is full. The join process takes the following steps: first, a new node selects a random address. Second, based on the address assigned, the node finds its near neighbor on both a caching network and a querying network. Third, one of the near neighbors informs a box whose boundary includes the newly assigned address. The box information is described by the boundaries of the box in terms of column and row addresses. The column elements of the address as well as the
row elements must be between the boundaries. Fourth, if the number of nodes in the box is not over the maximum number of nodes allowed in a box, the box is selected as the new node’s box. If the box is already full, the node sends a message to find a box containing the minimum number of nodes by bounded broadcasting. The selection of the range for the bounded broadcasting is described in Section 6.2. Upon finding a not fully occupied box, the new node changes its box selection to the box. If every box in a column or a row is full with the maximum number of nodes, then some box has to be split. Once the split process is called, all the boxes in the same column (or row) will be split. The split process will be discussed in the following section in detail. In the splitting process, the new node identifies a box whose boundaries include the node; then, the box is connected to the new node. Fifth, the selected box from above steps decides a new address in a proper position in the box. Finally, a new node joins the network with the assigned address and makes connections to the near neighbors and a shortcut neighbor.

Finding a proper position in a box for a new node is important since it is efficient if each box always maintains nodes in a way to be in a ‘splittable’ phase. If there is only one node in a box (this case can be seen only when there is one node in a whole network or a node fails due to the network dynamics), the box selects a diagonal position from a sub-box with the maximum number of nodes. By placing a new node in a diagonal position, the box always splittable either through column or row. The case of three nodes in a box also selects the same position for a new node to balance the positions of the nodes in each sub-box. In all other cases, a position is selected in a sub-box with the minimum number of nodes. Figures 6-2 through 6-5 show how a box chooses a proper position. In these example cases, the maximum number of nodes per box is set to 5. The box maintains the number of nodes in any sub-box. In Figure 6-2, a proper position for a new node must be sub-box 1 which is diagonal from sub-box 4. In Figure 6-3, either sub-box 2 or 3 can be selected. When at least 3 sub-boxes are
occupied by a node as shown in Figure 6-4 and 6-5, a new address in the sub-box with
the minimum nodes will be selected.

6.2.2 Split

When all the boxes in either a column or a row are filled with the maximum number
of nodes, the column or the row needs to be split. The first step of splitting is to check
if the boxes in the column (or row) are splittable or not. If there are at least two nodes
in the box and the two nodes are in the diagonal sub-box to each other, then the box is
splittable. If all the boxes are splittable, the node sends a ‘split’ message over the entire
column (or row). Note that nodes are evenly distributed over four positions in a box since
the join process places new nodes in that way. After splitting, each box still includes at
least the minimum number of nodes. Finally, a new node finds a proper position in the
split box. Figure 6-6 shows before and after splitting boxes in a row.

The selection of splitting direction (through the column or through the row) is
determined by comparing the width of column with the width of row of the box. If
\( \frac{w_c}{w_r} < 1 \), then the row will be split. The column will be split if \( \frac{w_c}{w_r} > 1 \). In the case of
\( \frac{w_c}{w_r} = 1 \), the algorithm selects either a column or a row to be split. By splitting through
the wider direction, the algorithm maintains each box as square as possible. The box
shape is closely related to the trade-off between the caching cost and the querying
cost. The square box, whose width of column and that of row is the same, is optimal in
terms of trade-off between the caching cost and the querying cost. Users can control
the shape of box by giving a priority to split in a certain direction. For example, narrower
width of column makes the caching cost lower, but one has to pay more querying cost,
and vice versa. If the widths are the same, the algorithm selects any direction uniformly
at random.

6.2.3 Trading-off Cache/Query Costs

In the algorithm and the simulation setting described in the previous sections for
Exact Deetoo, a symmetric address space, which is \( w_c = w_r = \sqrt{B} \) is assumed,
where \( w_r \) is the width of rows for querying network, \( w_c \) is the width of columns for caching network, and \( B \) is total address space. In the symmetric space, the number of columns and the number of rows are the same, and this incurs the same amount of communication cost for caching and querying. One can imagine the case of asymmetric caching and querying cost. For example, the space can be a rectangle with \( w_r \approx k w_c \), where \( k \) is some real number instead of 1.0. Figure 6-7 illustrates the example case of an asymmetric address space. In the asymmetric matrix, the condition for splitting is slightly different from that of a symmetric matrix. If \( \frac{w_c}{w_r} > k \), the algorithm splits a row in half, and it splits a column if \( \frac{w_r}{w_c} < k \). The split algorithm selects a row or a column randomly in case of \( \frac{w_r}{w_c} = k \). The case of more columns requires less caching cost but more querying cost, and the exact opposite in the case of more rows. Thus, Exact Deetoo provides more options for the network setting according to the desirable publishing and searching performance. In either case, Exact Deetoo always supports exact response by maintaining at least one node in each box.

### 6.3 Simulation Results

For the simulation of Exact Deetoo, I expanded Netmodeler from the Deetoo simulation. Five performance metrics are used: (1) Box properties - the number of boxes and the number of nodes per box as the network size grows. (2) Join cost - the number of generated messages for a new node join. (3) Caching and querying cost - the number of messages broadcast for replicating data and for searching data. (4) Load balance - the number of objects per node and the number of replica per unique object. (5) Latency - the depth of broadcasting tree. For measuring latency, I assumed that the latency between any two nodes are the identical.

In the simulation model, the minimum number of nodes per box is set to 2 and the maximum number of nodes per box is set to 5. Each box is occupied by at least the minimum nodes. The topology of simulation network followed one dimensional small world model as in Deetoo simulation. Each node forms two virtual rings, one for the
cache and the other for the query. In addition to near neighbors, each node makes a connection to a shortcut node according to the inverse $r^{th}$ power distribution. Except the join process including splitting algorithm, the network structure and routing algorithm are the same as those of Deetoo. While Deetoo routes a message (either a caching message or a querying message) over multiple number of columns or rows based on user defined replication factor, Exact Deetoo only needed to select a random node for a starting point for bounded broadcasting. A box associated with the node decides the range of bounded broadcasting based on the boundaries of the box. The size of the address space is set to 32-bit which has $2^{32}$ possible addresses. The notations used in the following sections are: $B_u(box)$ is upper bound of the number of nodes in a box, $B_l(box)$ is lower bound of the number of nodes in a box, $B_u(cost)$ is upper bound of the communication cost, $B_l(cost)$ is lower bound of the communication cost, $M$ is the maximum number of nodes per box, $m$ is the minimum nodes in a box, $K_c$ is the cost for a cache, $K_q$ is the cost for a query, $\bar{N}_{box}$ is the average number of nodes per box, $w$ is the width of a column/row, and $N$ is the number of nodes in the network.

### 6.3.1 Box Properties

The Figure 6-8 shows the actual number of boxes in a network as network size grows and the results are compared to the upper bound and the lower bound. The upper bound is achieved if all the boxes contain the minimum number of nodes. If all the boxes are full with the maximum number of nodes, the number of boxes reaches the lower bound. Thus,

$$B_u = \frac{N}{m} \quad (6-1)$$

$$B_l = \frac{N}{M} \quad (6-2)$$

I observed two phases in the increment of the number of boxes. One phase is a burst increment of the number of boxes that is called ‘splitting’ phase, and the other is called ‘filling boxes’ phase where the number of boxes is not changed. The figure 6-9
amplifies the two separate phases in log scale. When the number of boxes reaches $2^k$, where $k$ is any integer in $[0, \infty)$, the number of boxes stays constant until almost every box is full in the entire system. In the ‘splitting’ phase that is between two subsequent ‘filling boxes’ phases, the number of boxes increases. At the time that the number of boxes reaches $2^k$-power, the entire space is split evenly with the same number of columns and rows and makes each box in a square form, At this time of point, it is highly probable that each new node finds a box with the number of nodes under the maximum without demanding splitting. The join algorithm of the Exact Deetoo maintains ‘filling’ phase until almost every box reaches its maximum capacity. The ‘splitting’ phase starts after all the boxes is filled with the maximum. Since a new address is randomly selected at the first step of join algorithm, the joining node finds a fully loaded box with high probabilities even if some columns (or rows) are already split. The ‘splitting’ phase continues until all the column and the row are completely split Figure 6-10 shows the comparison of the number of boxes and the number of boxes with the maximum nodes. During the ‘filling a box’ phase, the number of boxes with the maximum increase linearly while the number decreases in the phase of ‘splitting’. The average number of nodes per box repeats a pattern of increment in ‘filling boxes’ phase and decrement in ‘splitting’ phase as shown in Figure 6-11.

6.3.2 Join Cost

The number of boxes in a network and the average number of nodes in a box determine the cost for joins. In the ‘splitting’ phase, join cost increases linearly since many node joins in ‘splitting’ phase requires bounded broadcasting to find a box with minimum or to split the column. On the contrary, join cost is decreasing in the ‘filling boxes’ phase. In ‘filling’ phase, many of join process finds a box where the number of nodes are below the maximum, thus the joining node does not need to bounded broadcast a join request message. I claim that the join cost is bounded by $O(\sqrt{N})$. There are 3 possible cases to confront when a node tries to join a network. In the
following analysis, I assumed that \( w = w_c = w_r \). For each case, the join cost is calculated:

**Case 1:** The box is not full. In this case, the node joins with the minimum joining cost. The node only needs to contact its neighboring nodes to identify a box whose boundaries include the node. It is required that neighbors in both a caching network and a querying network since the identified box from neighbors in one network cannot include the joining node’s address. The joining cost for this case is at most \( k \), where \( k \) is the number of near neighbors in both a caching and a querying network. In the simulation, the cost for this case is at most 4 (2 for each network).

**Case 2:** The box is full but there is at least one box that is not full in a selected column or row. In this case the new node sends a request for the box with minimum nodes using bounded broadcasting within the selected column or row. The cost for recognizing the box is \( \tilde{N}_{\text{box}} w \).

**Case 3:** All the boxes in a selected column or row are full. The case requires splitting all the boxes in the selected column or row. The costs for splitting is again \( \tilde{N}_{\text{box}} w \). The total cost for the case is the sum of the costs in case 1, case 2, and the cost for splitting, which is \( k + 2\tilde{N}_{\text{box}} w \). Thus, this case requires approximately twice as the cost of case 2. Since \( \tilde{N}_w = k \sqrt{N} \), where \( k \) is some real number, the cost is bounded by \( O(\sqrt{N}) \).

From the above analysis, I confirmed that the join cost is bounded by \( O(\sqrt{N}) \).

The phase transition between ‘filling’ and ‘splitting’ is observed in the Figure 6-12. The Figure 6-13 illustrates cumulative average of join cost The average cost stays \( O(\sqrt{N}) \) as claimed.

### 6.3.3 Caching and Querying Cost

For the test of communication cost for caching and querying, 100 string objects are inserted and each object is queried. Each insertion and querying is executed in a randomly chosen node. The resulting cost is averaged from each trial. In the Figure 6-14
and the Figure 6-15, the caching cost and the querying cost are compared to the upper bound and the lower bound. The bounds are calculated as following:

\[ m \leq \tilde{N}_{\text{box}} \leq M \quad (6-3) \]

Since \( \tilde{N}_{\text{box}} = \frac{N}{w^r} \),

\[ \frac{N}{M} \leq w^2 \leq \frac{N}{m} \quad (6-4) \]

\[ \sqrt{\frac{N}{M}} \leq w \leq \sqrt{\frac{N}{m}} \quad (6-5) \]

\[ \tilde{N}_{\text{box}} \sqrt{\frac{N}{M}} \leq w \tilde{N}_{\text{box}} \leq \tilde{N}_{\text{box}} \sqrt{\frac{N}{m}} \quad (6-6) \]

The cost is \( K_c = w \tilde{N}_{\text{box}} \).

\[ m \sqrt{\frac{N}{M}} \leq K_c \leq M \sqrt{\frac{N}{m}} \quad (6-7) \]

Since the symmetric space, where \( w_c \mid w_r \), the bounds for a query are the same as those for a cache. In other words, since the address space is defined as a square form and the split occurs in column and row alternatively, the number columns and the number of rows are almost the same at any certain point of time. This results in the same pattern of cost for both caching and querying. Similar to the above results, the alternating pattern is observed according to the change of box phases. In ‘filling boxes’ phase, the cost increases linearly while the cost has high variance in ‘splitting’ phase.

### 6.3.4 Load Balancing

Figure 6-16 shows the number of replica per unique objects in a network with 50,000 nodes. For the test, 500 unique objects were inserted and replicated. The number of replicas must match the caching cost \( K_c = w \tilde{N}_{\text{box}} \). Since the width of column/row is approximately 128 for the size of 50,000 network and the average number of nodes per box is 3.05, the expected cost (replication) is 390.4. The average number of replicated objects is 397.14 and the standard deviation is 64.64. Note that there are two separate distributions since the size of 50,000 network is in ‘splitting’ phase. There

105
exist two kinds of columns. The caching cost for the column that is already split is less than the cost for the column that is not yet split. With less then 10% of replication for this network size, Exact Deetoo always succeeds any query. In case of 20000 size of network which is in ‘filling’ phase, only one distribution of replication is observed in such a network as shown in Figure 6-17. In this size of network, the expected replication size is 294 since \( w \approx 69.74 \) and \( \tilde{N}_{\text{box}} = 4.11 \). The measured average is 304.60 and standard deviation is 18.73.

The number of objects per node tells how well the load is distributed. Figure 6-18 displays the distribution of the number of objects in each node in the 50,000 size of network. The number of objects per node is calculated as follows:

\[
\bar{m} = \frac{K}{m} \quad \text{(6–8)}
\]

, where \( \bar{m} \) is the number of objects per node. Thus, the expected number of objects per node is 3.90. The average and standard deviation of number of objects per node 3.97 and 2.14, respectively. Figure 6-19 shows the number of objects per node in the 20,000 size of network. The calculation is 7.19 objects per node and the measured average is 7.62 and standard deviation is 2.73.

6.3.5 Latency

Latency was measured by counting the depth of bounded-broadcasting tree. The result is shown in Figure 6-20 The results was similar to the results of Deetoo since Exact Deetoo utilize the same topology and routing algorithm.

6.3.6 Query Result

The Figure 6-21 illustrates the number of nodes that returns matching result for queries. The number of the nodes equals to the number of nodes in the box that visited by broadcast both for caching and querying. The pattern of increment and decrement repeated according to the shifting of box phases (filling, splitting). Note that any single
query is successfully resolved by at least a node. The result verified the search ability of the Exact Deetoo.

6.4 Conclusion

In this chapter, I presented Exact Deetoo. The main advantage using Exact Deetoo is that each and every query is resolved without sacrificing bandwidth usage. Exact Deetoo is self-organizing in that the system finds a proper place to join for a new node. The scalability of Exact Deetoo is validated by the simulation with a million nodes. The message overhead either for caching or querying is bounded by $O(\sqrt{N})$. Exact Deetoo is applicable to any ring-based structured overlay.
Figure 6-1. Example 2-dimensional 4-by-4 space
Figure 6-2. One node in the box

Figure 6-3. Two nodes in the box

Figure 6-4. Three nodes in the box

Figure 6-5. Four nodes in the box
Figure 6-6. Split boxes in a row

(a) Before splitting

(b) After splitting

Figure 6-7. Examples of asymmetric matrix space

(a) Cache-Friendly Matrix (Wc = 2Wr)

(b) Query-Friendly Matrix (Wr = 2Wc)
Figure 6-8. The number of boxes

Figure 6-9. The number of boxes in log scale
Figure 6-10. The number of boxes and the number of full boxes

Figure 6-11. The average number of nodes per box
Figure 6-12. Join cost (N = 100,000)

Figure 6-13. Join cost (N = 100,000)
Figure 6-14. Caching cost

Figure 6-15. Querying cost
Figure 6-16. The number of replication per unique object (50k nodes)

Figure 6-17. The number of replication per unique object (20k nodes)
Figure 6-18. The number of objects per node (50k nodes)

Figure 6-19. The number of objects per node (20k nodes)
Figure 6-20. Latency (tree depth)

Figure 6-21. The number of responding nodes
CHAPTER 7
CONCLUSION

Studying distributed network and data structure is of growing interest in computer networking research. The problem that P2P addresses is how to build reliable, large-scale systems without a high cost or management load on any one individual or organization. This dissertation focuses on P2P design approaches to make developing a new P2P systems easy. I presented the algorithms for developing P2P general search systems and the problems one can face for general search on top of P2P overlays. P2P applications can be developed by reusing the P2P search components. I proposed solutions for some of the problems.

In chapter 3, I proposed four different methods for network size in P2P network. The methods estimate the number of nodes based on the node density in a given length of address space. I also suggested a new joining method to improve the uniformity of node distribution. Each estimation methods achieves various performance and communication cost. A user can select a method or combined methods with respect to his/her desired performance. I compared each method by simulation in terms of the accuracy of size estimation and the communication cost.

In chapter 4, I introduced Deetoo, a scalable routing peer-to-peer protocol for unstructured networks which provides efficient caching and lookup functionality with a constant query hit-rate (without regard to a network size), \( O(\sqrt{N}) \) replication, \( O(\sqrt{N}) \) query cost, and \( O(\log^2 N) \) search time. Deetoo, allows objects to be updated and deleted, and can be used to handle general queries. Specifically, any problem that can be mapped onto selecting (with high probability) objects which match some query can be run over Deetoo.

Deetoo is deployed and tested on Planet-Lab nodes in Chapter 5. The implementation on real network, such as Planet-Lab can be meaningful. The real implementation requires writing an API to make it easy for people make new query and cache objects.
An API provides a good tool for people to use Deetoo in many applications. Deetoo’s routing algorithm is serviced on top of Brunet P2P substrate, then nodes with Deetoo functionality deployed on Planet-Lab nodes. The test results show the very close match to the simulation results as well as theoretical analysis. By investigating the performance of Deetoo on the real P2P environment, I assure that Deetoo is applicable query algorithm on top of many existing P2P overlays.

In chapter 6, Exact Deetoo is introduced and evaluated by simulation. Exact Deetoo is designed to resolve every query. The join cost, which is $O(\sqrt{N})$, is higher than randomized Deetoo. However, the communication cost for cache/query is still $O(\sqrt{N})$. In the simulation, I found that the number of box and the average number of nodes per box are the most important metrics for both join cost and communication cost.

Exact Deetoo will be applicable to many existing P2P overlays to achieve 100% query hit-rate. I believe that Exact Deetoo suggests a new direction of P2P data storage.
REFERENCES


BIOGRAPHICAL SKETCH

Tae Woong Choi earned a B.E. in the electrical engineering department at Yonsei University, Seoul, Korea in 1999. Approximately for three and half years, he worked as an engineer at Hanaro Telecom Inc. in the field of HFC (Hybrid Fiber Coaxial) network after his graduation. In 2003, he earned a M.S. in electrical and computer engineering at University of Florida in 2005 and he is continuing his academic experience to earn his Ph.D. in Electrical and Computer Engineering. He joined Advanced Computing and Information Systems Lab in 2005. His research interests include the area of P2P networks building components that can be put together to form novel P2P systems. He is working on an efficient network size estimation algorithms in the purely distributed networks. He is also working on modeling and evaluating scalable and efficient search systems. The search method will be applicable to any type of search including meta data search, partial matching, regular expression search, and other many general searches.