Efficient Peer-to-Peer Keyword Search Using Adaptive Space Partition

Chen, Po-Hung
Adviser: Tsay, Yih-Kuen
Department of Information Management
National Taiwan University

June 24, 2004

THESIS ABSTRACT
GRADUATE INSTITUTE OF INFORMATION MANAGEMENT
NATIONAL TAIWAN UNIVERSITY

Student: Chen, Po-Hung Month/Year: June, 2004
Advisor: Tsay, Yih-Kuen

Efficient Peer-to-Peer Keyword Search Using Adaptive Space Partition

Locating the node that stores a particular data item is a fundamental problem in peer-to-peer systems and is traditionally formulated as the construction of a distributed hash table (DHT). Typical DHTs support only a simple map from keys to values, but not keyword search. To support keyword search on a DHT, some distributed inverted index is needed. The challenge is how to evenly distribute the inverted index entries over the peers of the network. Because the keywords frequency in data items is usually of a zipf-distribution, simply partitioning inverted index entries by keywords would cause unbalanced loads. Furthermore, when there are more than one keywords in a query, some intermediate data have to be transmitted across the network and joined with each other to get the final result, incuring much network traffic overhead.

We propose a distributed indexing scheme, called Adaptive Space Partition (ASP), that has a good load balancing and incurs little network traffic overhead. The ASP scheme is designed to work on top of the CAN DHT. A CAN network is structured as a d-dimension virtual coordinate space, where each peer node is assigned to a zone in the space. The ASP scheme maps a keyword to a region, consisting of one or more zones. Each object related to a given keyword is inserted into a peer randomly selected from the keyword's region. To lookup objects related to the same keyword, a peer is randomly selected from the same region as a starting point, which then searches its neighborhood by flooding. Because objects with the same keyword are inserted into the same region, objects have good keyword locality and can be found easily. Furthermore, our scheme partitions inverted index entries according to the number of peers in a region to achieve better scalability. The ASP scheme also optimizes the query operation to have the same complexity as a CAN routing procedure.

Keywords: peer-to-peer, keyword search, DHT, overlay network

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

File sharing has long been the most popular peer-to-peer application. Popular peer-to-peer systems such as Freenet [1], eMule[2], eDonkey[3] usually has hundreds of thousands of users online at least. Suppose each user contributes a few files, then there will be millions of files available in the network. This is why peer-to-peer applications are so popular. It also means peer-to-peer applicatoins have to face a problem: how to find relevant objects from millions of files distributed on the peer-to-peer network?

There are three classes of peer-to-peer networks currently in use:

- Centralized Architecture: In this approach; some centralized servers are responsible for maintaining the index of all objects in the network. Every time when a peer wants to share an object, one or more entries are inserted to the central index server. The central server maintains a list of all the shared files in network. The query operation process as fellows: peer sends some search criterion to the central server, then the server scans the list and return matching objects. Napster is one example of centralized system. This kind of system is efficient when the network size is small, however the centralized server is a single failure point, which may suffer from the denial of service (DOS) attacks. For more, as the online peers become more and more, centralized approach does not scale well.

- Distributed Unstructured Architecture: This approach is on the extremely opposite of the first one. For example, Gnutella organizes peers into an unstructured overlay

network. There is no central index server exists. Peers use mass of unstructured links to communicate with each other. To search objects in Gnutella, a peer first sends a broadcast query message with a non-zero TTL to all its known peers. Since Gnutella use no central mechanism, no single failure point exists. It works very efficiently when the number of peers is few. However this flooding-style BFS mechanism doesnot scale well either. When the network grows, higher TTL is needed for the search operation to search the network, which creates large transmission overhead.

- Distributed Structured Architecture: This kind of architectures is also called as Distributed Hash Tables(DHTs). The DHTs such as PRR [4], Tapstry [5] , CAN [6], and Chord [7] emphasize on the scalability and reliability of the overlay network. In the DHT schemes, each object has a unique key (object ID). Given an object ID, the DHT algorithm routes the request to the peer, which is responsible for maintaining the reference of the desired object. The DHTs successfully overcome the scalibility problem. Unfortunately, DHTs dont support keyword search. It is a inherence of hash function that even a single different bit leads to a totally different object ID. As a result, no fuzzy search is allowed in a DHT,

To implement keyword search on a DHT, some inverted index is required. An inverted index is a data structure that maps keywords to object references.
For example:

keyword	object description	object reference
Music	Madana 2002 *Music*	p2p://www.xx.org/xx.mp3
	Music-Autumn leaf	p2p://www.xx.org/oo.wav
Guitar	Heavy metal *guitar* MTV	p2p://www.oo.org/xxx.mpg
	...	...

The challenge is how to design the right algorithm to evenly distribute the inverted index. One naive approach is to equally partition the DHT name space for each keyword. This approach is impractical because the keywords frequency in queries is a zipf-distribution [8]. Simply partitioning inverted index entries by keyword make some unfortunate peers responsible for much higher load, and thus make the system load unbalanced. A simple

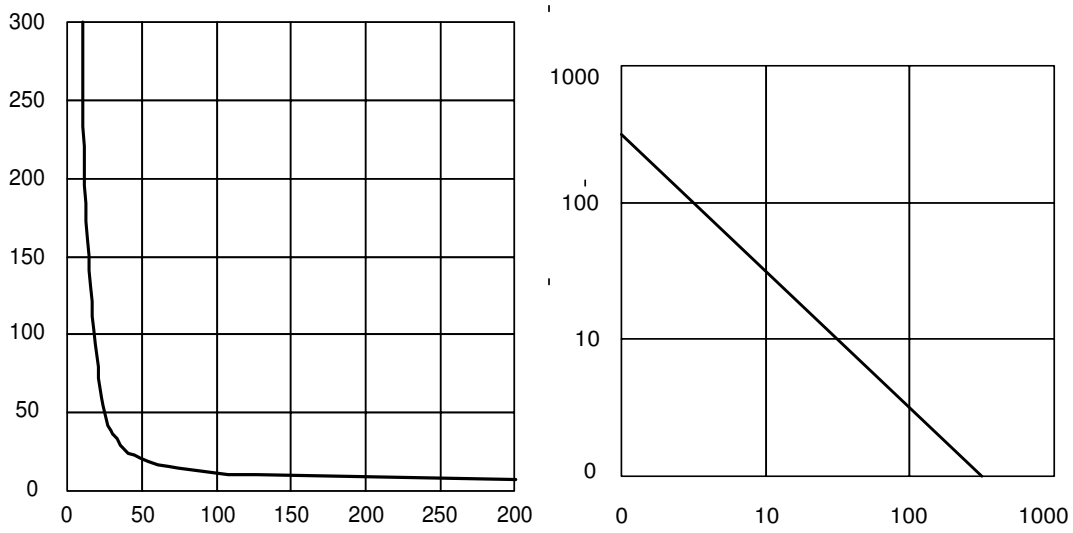

Figure 1.1: zipf-distribution

description of data that follow a zipf-distribution is that they have

- A few elements that score very high (the left tail in the diagrams)
- A medium number of elements with middle-of-the-road scores (the middle part of the diagram)
- Huge number of elements that score very low (the right tail in the diagram)

If the inverted index entries are partitioned by keyword, when the query uses more than one keywords, some intermediate data have to be transferred across the network and joined each other to get the final result. This join operation induces very large transmission overhead, which we do not want. To overcome this problem, Patrick Reynolds and Amin Vahdat have proposed a scheme using bloom filters. Bloom filters do greatly reduce the transmission overhead, however this approach still suffers from the unbalanced loads.

The DHTs are designed for large-scale peer-to-peer overlay networks, but when the number of peers is small, most DHTs cannot work very well. Some experimental results show that when the number of peers is small, simple approaches such as centralize servers or message flooding work much better.

Keyword search is the most natural approach for people to find desired information, but current peer-to-peer keyword search schemes do not do it very well. We are motivated to

develop a new scheme. We expect this scheme to be adaptive, efficient, and avoid undesired characteristics mentioned above, which exists in current peer-to-peer keyword search systems.

1.2 Objectives

We believe that the query operation is the most critical of a peer-to-peer resource sharing system, and the end-user latency is the most import performance measure. Because users search resource much often than sharing. We want to optimize our algorithm for the query operation.

Current DHT designs offer many good properties such as efficiency, fault tolerance, and the most important of all: scalibility. However, as mentioned above, the DHT is designed for large-scale networks and is not as efficient as the flooding-style or centralized system when the number of peers is small. To overcome this problem, we design the ASP to be a hybrid system, such that when peers are few, it tends to become like a flooding-style system; and when the number of peers increases, it become much like a DHT.

Since inverted index entries are partitioned by keyword, when there are more than one keywords, some join operations have to be performed. We want to minimize this kind of overhead. If much of the work is done locally, no join cost is required. Our approach is a trade-off between transmission cost and storage. In current networks, bandwidth is shared by everyone and still one of the most limited resource. On the other hand, a current desktop always has very big storage whose utilization is often low. So we think it is reasonable to trade storage with bandwidth, and this approach would achieve better system performance.

The ASP scheme uses a regular expression filter and dynamic space partition; thus for each query, whether it uses one keyword or more, only a constant number of messages are require.

1.3 Thesis Outline

In Chapter 2, we survey related works. These related works includes structured overlay network like CAN, unstructured overlay networks such as Gnutella and Napster, and other schemes, which are built on top of them to support more complex functionality.

In Chapter 3, we presents our ASP secheme, including the main idea of ASP, basic algorithm about object insertion and query; and the most important of all: how the adptive space partition works.

In Chapter 4, we describe our experimental model and performance evaluations such as object insertion overhead, query overhead and the peer load distribution.

Finally, we present conclusions and future work in chapter 5.

Chapter 2

Related Work

Recently a number of systems have been developed to support keyword search on peer-to-peer systems. In this chapter, we first introduce the Content Addressable Network. CAN is a DHT algorithm, which scales well but does not support keyword search. We then introduce Gnutella nad Napster. Gnutella is a fully distributed, flooding-based system, and Napster is a centralized system. Their designs are totally different but both suffer from the scalibility problem. Finally we introduce several systems, who tries to build the keyword search functionality on top of overlay networks.

2.1 Content Addressable Network (CAN)

CAN [6] is a distributed hash table, the basic operations of CAN include insertion, lookup and deletion of (key, value) pairs. The CAN network is formed by many individual peers. Each peer occupies a zone in the CAN virtual space. Each shared object is inserted to one coordinate in the virtual space. The peer whose zone covers this coordinate is responsible for maintaining the reference of that object. Object lookup is done by invoking a CAN message routing procedure, this procedure routes the lookup request to the peer who has the reference of the desired object.

2.1.1 Routing in CAN

Routing in CAN works by sending the message directly from the source peer to the destination peer along strait lines in each Cartesian space dimension.

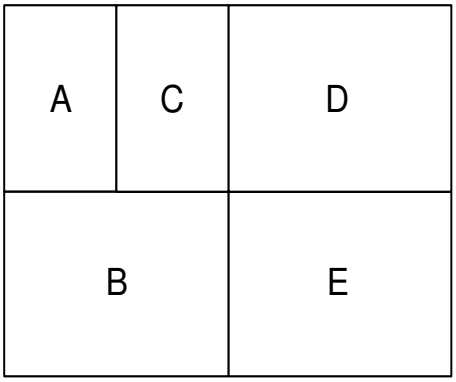

Figure 2.1: CAN virtual space

Each peer maintains its own routing table, which contains the network address and the virtual space of its immediate neighbors. In a d-dimension Cartesian space, two peers are neighbor if their space spans overlap along (d-1) dimensions and about along one dimension. In the following figure, the square stands for a 2-dimension CAN virtual space. In this example peer A is peer B's neighbor because A's zone is overlap with B's zone along the vertical dimension and about along the horizontal dimension.

A message routing includes the coordinates of the CAN virtual space. The destination peer is whose zone covers the target coordinate. The routing is done simply by greedy routs the message to its neighbor, whose zone is closest to the destination peer. Suppose the peers is uniform distribute to the CAN virtual space, the average routing length is $O(n\frac{1}{d})$ (d is the dimension of the CAN virtual space and n is the number of peers in the overlay network)

2.1.2 Network Construction in CAN

When a new peer joins CAN, it is assigned a zone, which it is responsible for. The zone allocation is dynamic. A new peer gets its zone by a existing peer splitting its own space, remaining half and handing the other half to the new peer.

There are three steps involve in the join procedur

- Find a peer that is already in the CAN.
- Using routing mechanisms, it would find a peer whose zone will be split.

- Finally, the neighbor of the split space must be notify so that future routings can include the new peer.

2.1.3 Zone Assignment

The new peer randomly chooses a point P in the space and sends a $JOIN$ message request destined for the point P. This message can start from any peer in the CAN overlay network. Then peer uses the CAN routing mechanism to forward the message to P. The message forwarding keeps on until the message reaches the peer in whose zone P lies.

The current occupant peer then splits its zone in half and keeps one half as its new zone and gives another to the new peer. The split must make the peer uniform distribute to the CAN virtual space. For example, in 2-dimension CAN, split is first split along the vertical dimension and at the next time horizontal dimension is selected and so on. When peer leaves, the zone is remerged. A leaved peer give back its zone to one of its neighbor. Some other DHT also use the dynamic virtaul space partition scheme like CAN, such as P-Gird [9].

Routing example:

1. The new-joining peer must find a peer already in the CAN network. In this example, the new peer first contacts with peer No.1.

2. The new peer selects coordinate (x, y) as its destination.

3. Then peer No.1 invokes the CAN routing procedure to send the $JOIN$ message.

4. Each peer greedy forwards the $JOIN$ message to the closest neighbor according the destination's coordinate.

5. Finally the message arrived peer No.7, then peer No.7 splits its zone to half and assigned one half to the new peer. (see the figure below)

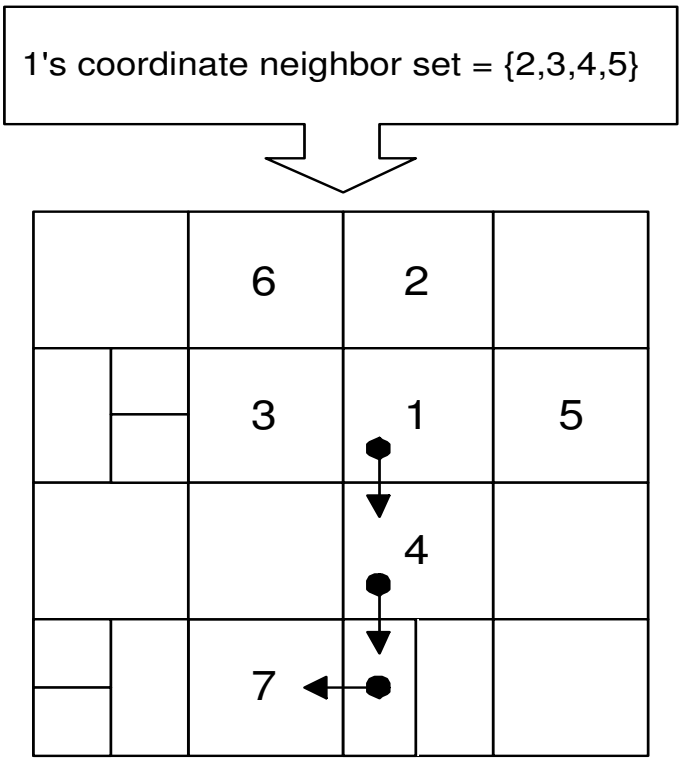

Figure 2.2: routing

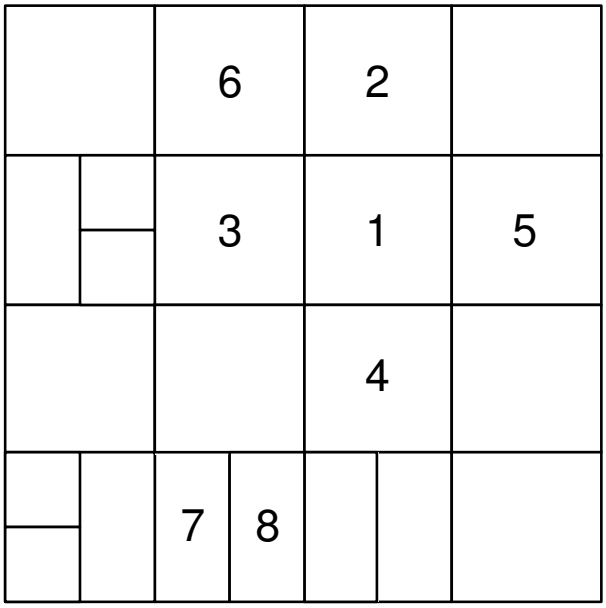

Figure 2.3: space partition

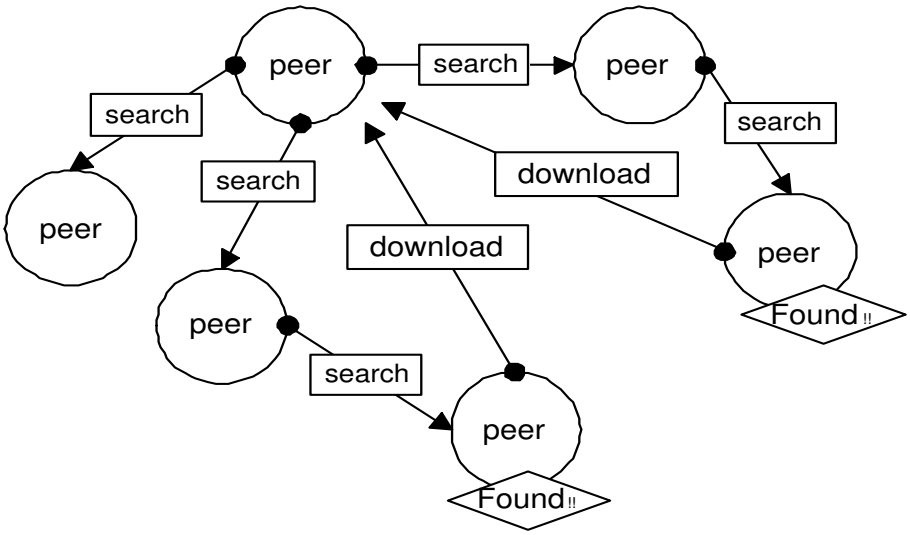

Figure 2.4: Gnutella

2.2 Gnutella

Gnutella [10] is a peer-to-peer protocol designed for file sharing. It works by broadcasting the queries with a given non-zero TTL to all the neighbors.

Since Gnutella is a unstructured overlay network, the querying peer does not know which peer might be sharing the desired file, all it can do is to broadcast the query to all it's neighbors. If the keyword is popular, then many matching objects can be found in a few levels. If the keyword is unpopular, peer has to set the TTL of the Query to a large number within the informal limit imposed by the clients in the Gnutella protocol. Thus, the Gnutella system causes a lot of network load during a query.

Current research [11], [12] shows that queries in Gnutella network also has a high locality.

Based on this high locality property, caching schemes [13] are proposed to further improve the Gnutella performance.

Unlike Gnutella, some unstructured peer-to-peer networks use a breadth first search instead of a depth first search [14], [15]. They also use some statistical scheme to improve system performance.

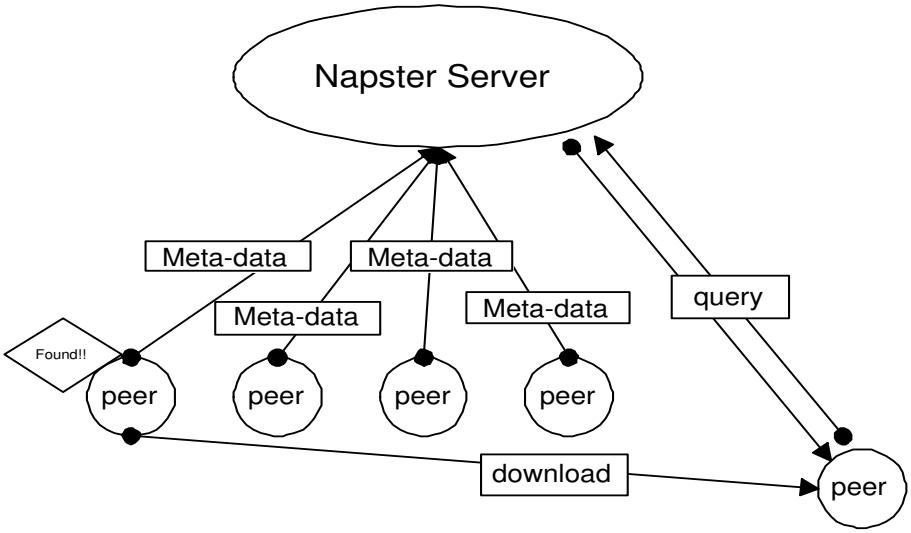

Figure 2.5: Napster

2.3 Napster

Napster [16] is a legend of the peer-to-peer application. It is designed for music sharing.

Actually Napster is not a pure peer-to-peer system. In Napster, each peer connects to a central Napster server. Napster server keeps the meta data of all objects. The indexing and searching is centralized. When a new peer joins the Napster system, the peer sends the metadata, which describes all the objects it has to the Napster server. Every search request is first sends to the Napster server. When receiving search request, the Napster server searches its database built with metadata sent by peers. The search result is a list of URL, which reference to desired objects. User then downloads the file directly from the peer according to the returning URL list. From the indexing perspective, Napster is a centralized system. Only files are transmitted from peer to peer.

2.4 Distributed Join Using Bloom Filters

A search for a single keyword consists of looking up the keyword's mapping in the index to get all of the objects containing that keyword. A "AND" query consists of looking up the sets for each keyword and returning the intersection. In traditional search engines, only a small subset of the matching documents is returned. Though in the distributed inverted

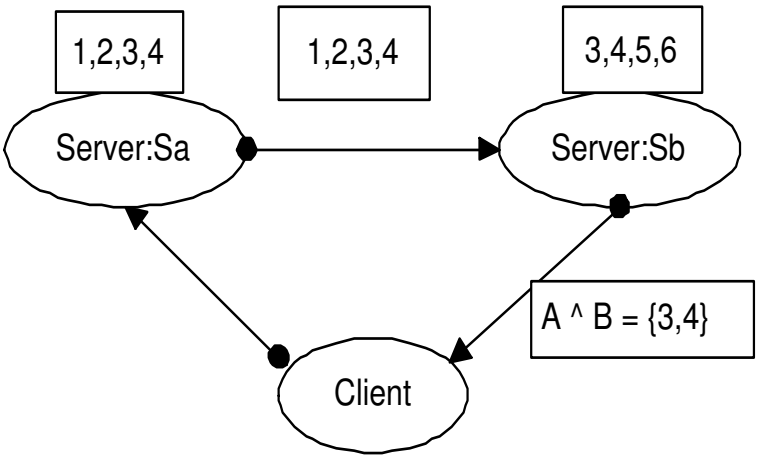

Figure 2.6: simple intersection

index system, this operation involves contacting several inverted index servers across the wide area, and each intersection consumes network bandwidth and latency. For example, client invokes a query "$A \cap B''$, and the inverted index of "A'' is maintained by Server S_a and the inverted index of "B'' is maintained by Server S_b. Client first contacts S_a and A sends the temporary result $1, 2, 3, 4$ to S_b. S_b then intersection $1, 2, 3, 4$ with its inverted index (inverted index about keyword "B''), and then send the result $3, 4$ back to client. This naive approach is not efficient because it sends large number of data across the network.

Patrick Reynolds and Amin Vahdat [17] proposed a scheme using bloom filter [18] to reduce the transmission overhead. Fore example, to compute the result of a query consisting of more than one keyword, the client first send its request to S_a. Then S_a sends a $F(A)$ ($F(A)$ is a Bloom Filter based on its list of inverted index for A) to a server S_b that contains the inverted index for the other keyword "B''. Upon receiving a bloom filter, S_b intersects the filter with its list of documents for "B'' and sends the intersection: $B \cap F(A)$, back to the node S_a. The node S_a then computes $A \cap B \cap F(A)$ which is the result for the query $A \cap B$.

Since the bloom filter is much smaller than the list itself. Sending bloom filter instead of index itself greatly reduce the transmission overhead. Ocean Store [19] also use bloom filter as its basic search scheme.

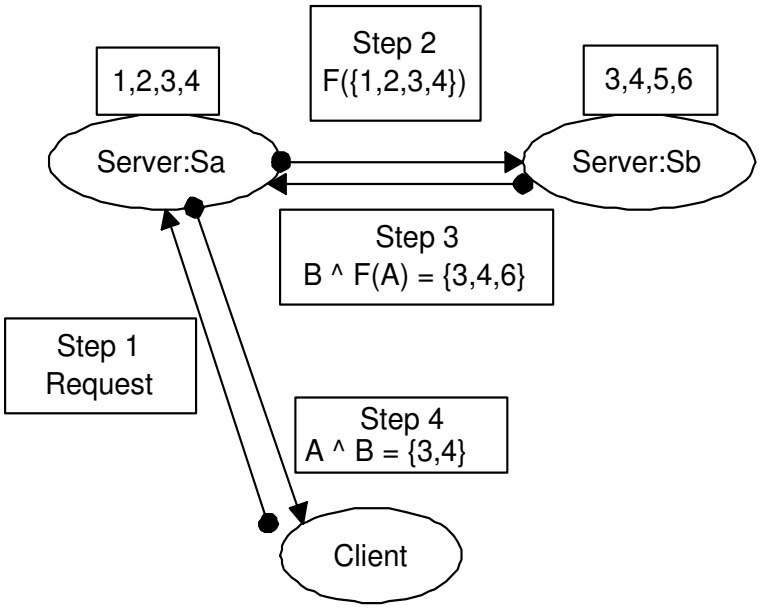

Figure 2.7: bloom filter intersection

2.5 Peer-to-Peer Prefix Search

Since the absence of keyword search is DHTs' characteristic, Baruch Awerbuch and Christian Scheideler propose a compromise solution [20]. It combines existing (non-searchable) distributed data structure in a transparent and consistent way to support the prefix search.

The upper layer is a chord-like DHT (such as SPRR [21], Skip list[22] or Chord[7]). Each object is inserted as one node in this chord-like DHT and ordered by its name. Because all objects in the network are linked according to lexicographic order, it's easy to support prefix search by routing search request along the chord-like ring until some objects is found.

The lower layer is responsible for organizing peers. It works like a typical DHT, which maps a key to an object. In this layer, a node stands for a peer in network. The lower layer linked all the online peers together and provides a communicate functionality. Using the DHT message routing mechanism, every peer can send message to any others easily.

The two DHT overlay works as fellows: Given a prefix, upper layer invokes a routing procedure to send a query message. This query message is sent to a node which covers the prefix.

When the routing is done, matching object's reference can be retrieved, and then the

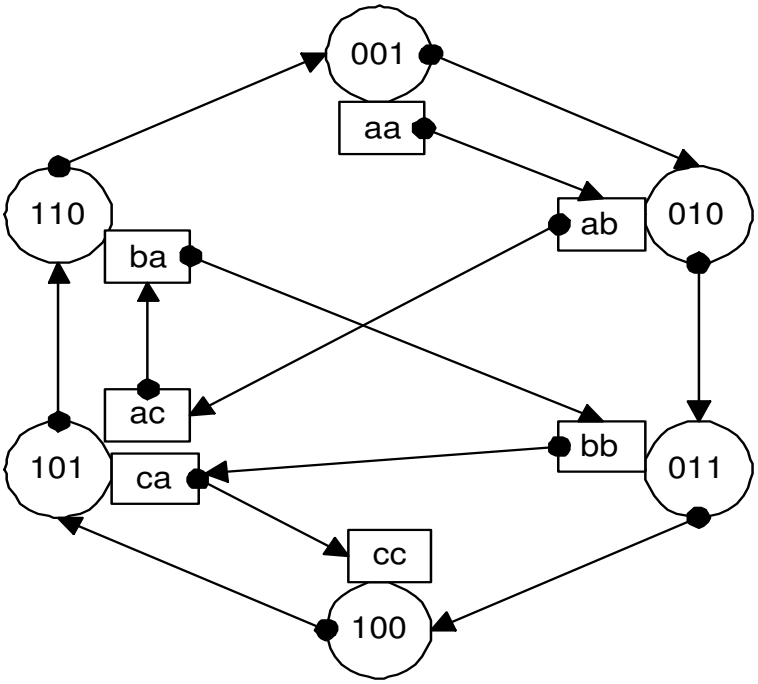

Figure 2.8: the PnP scheme

client can download desired objects.

Note the upper layer only maps a prefix to a virtual name space ($[0, 1]$), it knows nothing about physical network address. During each hop of upper layer's routing, another routing in lower layer is needed. The lower layer maps a virtual address ($[0, 1]$) to physical network address, so the upper layer can foward routing messages to next hop actually.

2.6 Keyword Set Search (KSS)

The keyword-set search (KSS) [23] system proposed by O. D. Gnawali works on a generalized DHT to support keyword search. The main idea of KSS is simple: since the DHT maps one ID to one object, if we bind every keyword set in a object's name to this object, then anyone of them could be used to locate the original object.

2.6.1 System Model

In KSS, an object o is given a keyword set Ko. In typical DHT, only the full object name is bind to the object. To support keyword search, every subset of Ko is bind to the object.

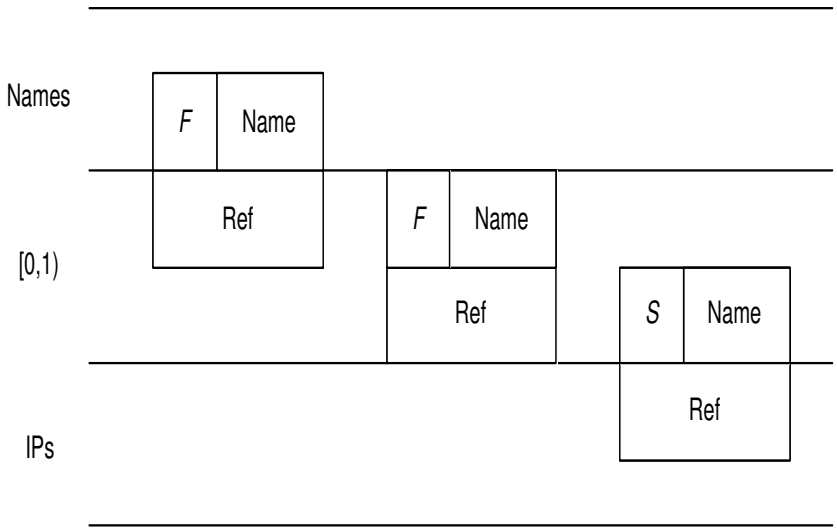

Figure 2.9: the layer stacks

Because most queries are short, it's not worth inserting every subset (the power set) of the Ko. In KSS there exists a globally known parameter S, which stands for the threshold of the keyword set length. Only sets whose length are not longer than S need to be inserted. In an extreme case, when S is bigger than any keyword sub-set length, all n^2 keyword sets (power set of Ko) are inserted. Currently KSS applications set S to be two, making an object indexed in $O(|Ko|^2)$ inverted index.

2.6.2 Insert and Delete

Suppose an object has a keyword set $K_o = kw_1, kw_2..., kw_n$ and $S = 2$. To insert this object, all sub-set: $K_i(i = 1\ n)$ and K_i, K_j (for all i, j — $1 <= i < j < n$) is inserted into the network. To delete the object, all involved peers are contacted and all those inverted index entries are removed.

2.6.3 Query

To resolve a query q, which has a keyword set $K_q = kw_1, kw_2..., kw_n$. First partition K_q into arbitrary mutual exclusive keyword sets, each of which is no longer than S (for example, $kw1, kw2, k1, k3...$), Use each of them as query key and the final result is the intersection of their results. KSS further improves the performance of query by piggyback the full K_o with

the keyword set in every inverted index entries. This makes peer able to provide the final result via a local lookup.

2.6.4 Storage overhead

If parameter S is set to be the infinite, making any possible combination of keywords an index, the insert operation will cost very much. Though KSS assume the infinite storage, in practical it is impossible to have infinite storage. To make a long story short, KSS trades the costs of storage for query efficiency.

2.6.5 Load Balancing

If the system are configured to use only length-2 keyword set, KSS actively prevents the hop spot problem. This is because queries containing a certain popular word k can be serviced by some other peers, which k ,* maps to rather than only by the peers, which k maps to. Though in this configuration, length-1 queries are not supported.

2.7 PeerSearch

Peer Search [24] builds IR models on top of the CAN overlay networks. They propose VSM (vector space model) and LSI (Latent semantic indexing) as the candidate IR models.

2.7.1 Vector Space Model (VSM)

In VSM [25], objects are represented as term vectors. Each element in the vector corresponds to the importance of a keyword in the object description or query. The importance of a keyword is often computed using statistical $TF * IDF$ (term frequency * inverse document frequency)scheme. Term frequency in the object and term frequency in other documents decide the importance of this term in an object. If a term appears in an object with a high frequency, there is a good chance that the term could be used to differentiate the document from others. However, if this term also appears in many other objects, its importance is low. In processing a query, a common measure of similarity is the cosine of the angle between vectors. Given two vector, $X = (x1, x2, ..., xl)$ and $Y = (y1, y2, ..., yl)$, the similarity be-

tween them is defined as $\cos(X, Y)$, which denotes the angle between vector X and Y. Note that X and Y are normalized, which means $|X| = 1$ and $|Y| = 1$. The similarity is simply the inner product of the two vectors.

$$\cos(X, Y) = \frac{X \odot Y}{|X| \cdot |Y|} = \Sigma X_i Y_j$$

2.7.2 Latent Semantic Indexing (LSI)

The VSM model suffers from noise in object. To solve the problem, the LSI [25] models uses statistically derived conceptual indices instead of terms for retrieval. It uses singular value decomposition (SVD) to transform a high-dimensional term vector (derived from VSM) into a lower-dimensional semantic vector. Projecting the vector into a semantic subspace does this transformation, and after the transformation, each element in semantic vector presents an importance of a concept in the object or query. LSI uses cosine of two vectors to presents similarity between two objects, the same with VSM's approach.

2.7.3 Basic Algorithm

P-LSI sets the dimensionality of the CAN to be equal to that of LSI's semantic space. Inverted index for objects are inserted in the CAN using its semantic vector. The inverted index includes the semantic vector of a document and a object reference to the object.

1. When receiving a new document A_a, the engine peer generates its semantic vector Va using LSI and uses it as the key to insert the inverted index entries in the CAN.

2. When receiving a query q, the engine peer generates its semantic vector Vq and routes the query using V_q.

3. Upon reaching the destination, the query is flooded to peer within a radius r, determined by the similarity threshold or the number of wanted documents specified by the user.

4. All peers that receive the query do a local search using LSI and reports the reference to the best matching documents back to the user.

2.8 Peer-to-Peer Keyword Search Using Keyword Relationship

The Keyword Relationship Search System [26] is a peer-to-peer keyword search system, which increases possibility of discovering desired objects. It's key mechanism is query expansion, which means adding relevant keywords to the original query. This query expansion is based on a keyword relation database (KRDB), which is managed in a distributed fashion by participating peers. The KRDBs is improved through search and retrieval processes and each relationship are shared among peers holding similar objects.

2.8.1 KRDB

Each peer maintains its own KRDB. In initial state, peer keeps information about keyword relevant only to objects stored locally. For each pair of relevant keywords, two entries are kept. The keyword relation: KR and its strength: $KRStr$. Keyword relation is transitive. For example, if the value of $KRStr(F, E)$ * $KRStr(E, I)$ is larger than a system-wide threshold, a relation between F and I is created.

2.8.2 Search

The query message is propagated in the same way as traditional unstructured peer-to-peer system. Peer received the query message appends some relevant keywords according to its own KRDB, and uses the expanded query to search it's local objects and returns matching objects to the searcher. Then if the message's TTL is still not reduced to zero, the peer forwards the *original query* to the next hop and so on so forth. Peer only forwards original query because consecutive query expansions lead to query explosion with less relevant keywords.

2.8.3 Distributed KRDB Updates

Because query expansions are done according to distributed KRDBs. The accuracy of KRDBs significantly affects search performance. Therefore KRDBs are required to be updated and kept as accurate as possible. Two mechanisms are used to updates the KRDB;

evaluation feedback and *KRDB synchronizations*.

The evaluation feedback works as fellows. Suppose a searcher initials a query contains keyword ki, and during the query procedure, keyword kj is used to expand the query and an object with keyword ki and keyword kj is found. If searcher selects this object, the relation $KR(Ki, Kj)$ is regarded as helpful and $KRStr(Ki, Kj)$ is increased, otherwise it is decreased instead. Keyword relations whose strength falls below a system-wide threshold are deleted from KRDB. $KRStr(Ki, Kj)$ is defined as:

$$KRStr(Ki, Kj) = \frac{HelpfulCnt(Ki, Kj)}{UsedCnt(Ki, Kj)}$$

The $UsedCnt$ records how many objects are found by expanding the query with relation $KR(Ki, Kj)$, and $HelpfulCnt(Ki, Kj)$ records how many times the found objects are regarded by searcher as helpful. During a query, peers involved in the procedure use feedback messages to notify each other and update their KRDB according to the equation listed above.

However, evaluation feedback has two drawbacks. First, evaluation feedback only evaluate existing KRs, which are extracted from local data items. Second it would take a long time to make the value of $KRStr$ statistically meaningful. For this reason, peers with short lifetime or the peers that have just begun to expose new objects cannot provide accurate KRDBs soon.

Another mechanism called KRDB synchronization is used to overcome these drawbacks. KRDB synchronization works as fellows. First peers use broadcasting message to find other peers, who have enough common keywords in their KRDB. Then they synchronize with each other. The synchronization procedure has two steps; (1) If one peer has KRs relevant to another peer, those KRs are added to another peer's DRDB. (2) If these two peers has some keywords relation in common, the one with bigger $UsedCnt$ are considered as statistically more correct, and they both update their $KRStre$ to the more correct one.

Chapter 3

Keyword Search Scheme

3.1 Resource Sharing in Peer-to-Peer Environments

Current research for peer-to-peer networks [12] show many characteristics different from traditional information retrieval systems. Some characteristics are listed below:

- Most shared objects are in binary format such as musics, movies or applications.
- Users look up resource much more often than share resource.
- Most objects are described by a short title which contains only a few keywords.
- Most queries contains no more than two keywords.
- The distribution of keyword popularity is mostly stable.
- Most users do not need an exhausted search.

Our goal is to utilize these characteristics to design a high performance peer-to-peer system.

3.2 Basic Idea

A CAN network is structured as a d-dimension virtual coordinate space, where each peer node is assigned to a zone in the space. The ASP scheme maps a keyword to a region, which consists of one or more zones. Each object related to a given keyword is inserted into a peer randomly selected from the keyword's region. To lookup objects related to the same

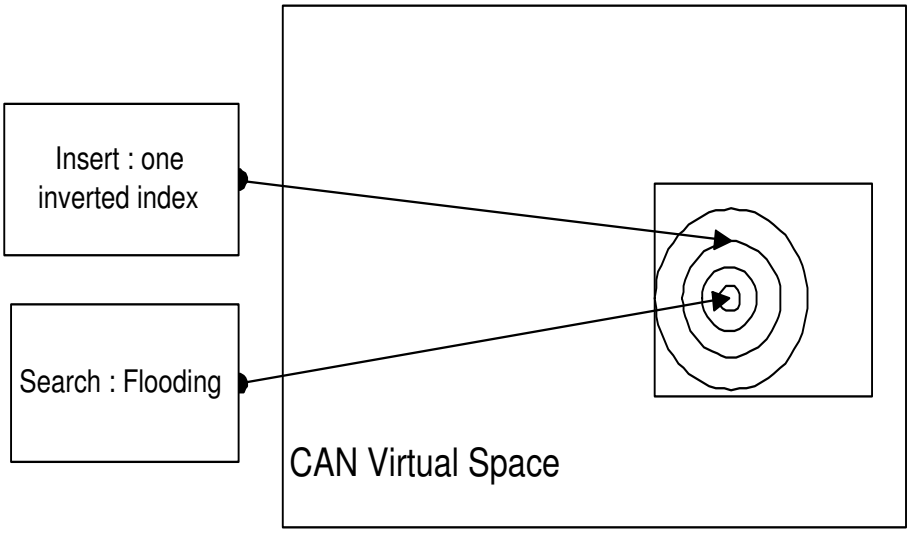

Figure 3.1: basic algorithm

keyword, a peer is randomly selected from the same region as a starting point, which then searches its neighborhood by flooding. Because objects with the same keyword are inserted into the same region, objects have good keyword locality and can be found easily.

3.3 Challenges

Although our scheme is simple, there are many issues we need to consider:

- Load balancing

  The object related to a keyword and the query related to a keyword is not uniformly distributed. How to assign regions to keywords and achieve load balancing is a big problem.

- Network Traffic Load

  Flooding messages cost a lot of network bandwidth. Stop conditions of flooding message greatly influence the system performance. Simply use a application defined TTL(time to live) as the flooding stop condition does not work well.

- Incremental Results

  The number of matching objects for any given keyword is roughly proportional to the

number of objects in the network. Thus, the cost of returning all matching objects to searcher grows linearly with the size of the network. Fortunately searchers rarely need all revelant objects in network. By using streaming transfers and returning only the desired number of results, we can greatly reduce the amount of data that needs to be sent. This is, in fact, critical for scalability:

- Query Performance

  Query is the most often operation in the peer-to-peer keyword search system, much more than object insertion. Good system designs trade insertion overhead to optimize the query efficiency.

- Robustness v.s. Scalability

  A flooding system is very robust, and is also efficient when the number of peers is small. On the other hand, a DHT system is scalable when the number of peers is large. Our system combines these two schemes in a dynamic way, thus when the network is sparse, it works like a flooding system, and when the network size grows up, it become more like a DHT.

3.4 Region Assignment

The region assigment scheme determines the load balancing of the system. The region of a given keyword is calculated with the keyword's frequency. So, peers need to know the frequency of each keyword in advance.
Assumption:

A peer can locally estimate keywords frequency both in objects and queries.

Those estimations do not need to be very accurate, but accurate estimations make the ASP system more efficient.

There are two ways to support this assumption. The first one is to use piggyback messages. In CAN, peers exchange messages periodically to maintain the network topology. Peers piggyback keyword statistics to those messages, and they also keeps all messages received from other peers whatever it is online or not. Those hitorical messages are used to

estimate keyword frequency. Another approach, as we explain later, is to use a globally unique profile. Each approach has advantages and disavantages.

Our goal is to distribute the keyword inverted index to the CAN virtual space according to its frequency to achieve the load balancing.

Peers need two statistics, which are list bellow:

- Keyword frequency for insertion - IFreq(keyword):

  occurrence of the given keyword in objects/
  number of words for all inserted objects

- Keyword frequency for query - QFreq(keyword):

  occurrence of the given keyword in queries /
  number of words for all queries

Frequency of a keyword is defined as fellows:

$$Freq(keyword) = \alpha \cdot IFreq(keyword) + (1 - \alpha) \cdot QFreq(keyword)$$

where $0.0 <= \alpha <= 1.0$

There is a hash function:

$$Hash(keyword) -> v = c_1, c_2, c_3 ... c_d$$

where d is the dimension of CAN, and for each $i \in 1, 2, 3...d, 0.0 <= c_i <= 1.0$ This hash function maps a keyword to a vector. Each element in the vector stands for a coordinate in one dimension. For example, c_i is a coordinate in dimension i.

Then we design a size function:

$$Size(keyword) = Freq(keyword)^{\frac{1}{d}} \cdot \beta$$

where β is simply a scale factor. The size function determines how long the region of that keyword crosses each dimension.

Finally, the Region function is defined as fellows:

$$
\begin{aligned}
&Region(keyword) = \{ \\
&(c_1 + \tfrac{1}{2} \cdot Size(keyword), c_1 - \tfrac{1}{2} \cdot Size(keyword)), \\
&(c_2 + \tfrac{1}{2} \cdot Size(keyword), c_2 - \tfrac{1}{2} \cdot Size(keyword)), \\
&\dots , \\
&(c_d + \tfrac{1}{2} \cdot Size(keyword), c_d - \tfrac{1}{2} \cdot Size(keyword)), \\
&\}
\end{aligned}
$$

The region is define so that

$$\frac{Region(keyword)}{total - space} = \beta \cdot Freq(keyword) \propto KeywordFrequency$$

Note the region of different key-word may overlapped.

3.5 ASP Interative Deepening

The ASP iterative deepening scheme is designed to decrease network load and support incremental results. We use a modified iterative deepening [27] called ASP Iterative Deepening as our flooding scheme. It works as fellows; Define MAX as a system parameter, which is the threshold of the number of flooded peers during a query. When the number of peers in a group is no more than MAX, they can communicate using flooding efficiently.

Because peers in CAN are uniformly distributed, peers can also use the piggyback message to estimate peer density . Define

$Npeers(region)$: the number of peers in a region

The ASP interative deepening works as follow:

1. Start peer broadcasts the search message to its neighbors, which is implied by B in figure.

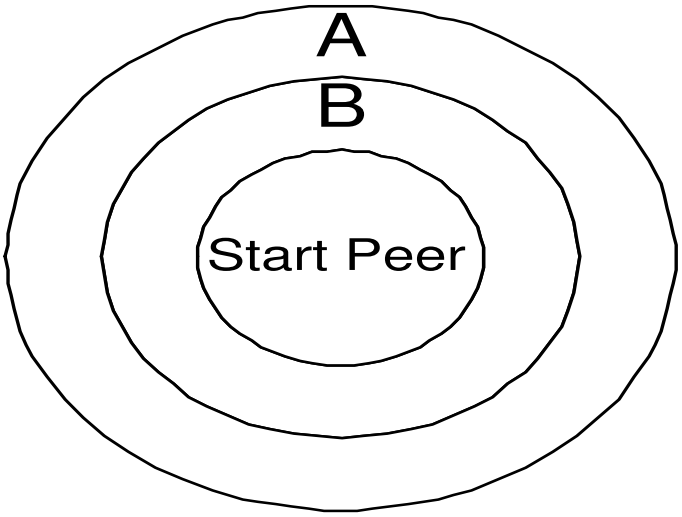

Figure 3.2: progressive flooding

2. Peers in B search its local inverted index entries and return matching ones.

3. Peers in B also return the peer addresses in A, which are neighbors of neighbors of start peer.

4. Start peer evaluates the number of the found objects. If its equal or more than $USERWANTED$, the flooding stops. If not, it sends broadcast messages to peers in A directly.

   This process continues until more than $USERWANTED$ objects have been found or the boundary of keyword region has been reaches. Three conditions stop the start peer forwarding flooding messages.

- More than $USERWANTED$ objects have been found.
- Neighbor peers is outsize of keyword's region.
- More than MAX peer has been flooded.

3.6 Basic Insertion

To insert an object, first stem each keyword in the object title to a normal form. In this procedure words whose frequency are bigger than a system parameter: $FREQMAX$ are

regarded as useless and are ignored. Then each *meaningful* keyword is stemmed to a normal form. Stemming is a process for removing the commoner morphological and inflexional endings from words in English. Its main use is as part of a term normalisation process that is usually done when setting up Information Retrieval systems. In this thesis, we use Porter's Stemming Algorithm [28] as our stemming scheme. Other stemming algorithms can be used too. After the stemming, inserte this object for each keyword in its title. For example:
Object : *" What a wonderful world!"*

- Step1: Filtering

Word[0]	what	*selected*
Word[1]	a	$Freq(\text{“}a'') > FREQMAX$, *Ignored!*
Word[2]	wonderful	*selected*
Word[3]	world!	*selected*

- Step2: Steming

Word[0]	what	
Word[1]	wonderful	stem to “*wonder*″
Word[2]	world!	stem to “*world*″

- Step3

  For each of the region: $region(\text{“}what'')$, $region(\text{“}wonder'')$, $region(\text{“}world'')$, randomly select one peer in the region and insert one inverted index entry in it. The peer selected in $region(\text{“}what'')$ keeps record like that:

prefix	object title
what	*what* a wonderful world!

  The peer selected in $region(\text{“}wonder'')$ keeps recorder like that:

prefix	object title
wonder	what a *wonderful* world!

The peer selected in $region(\text{“}world\text{”})$ keeps recorder like that:

prefix	object title
world	what a wonderful *world*!

3.7 Basic Lookup

Suppose the query string is: $\text{“}whata * wonderful*\text{”}$,

- Step1: Filtering

Word[0]	what	*selected*
Word[1]	a	$Freq(\text{“}a\text{”}) > FREQMAX$, *Ignored!*
Word[2]	wonderful	*selected*

- Step2: Steming

Word[0]	what	
Word[1]	wonderful	stem to $\text{“}wonder\text{”}$

- Step3: Expansion

  Extend the query to this form:

prefix	filter
what	what a * wonderful *
wonder	what a * wonderful *

- Step4: Request Routing

  The object: *“what a wonderful world!”* can be found using ether in $region(\text{“}what\text{”})$ or $region(\text{“}wonder\text{”})$, though search from the lower frequency keyword is more efficient.

Because keywords with lower frequency may have smaller region and have lower cost for flooding.

Suppose the keyword "$wonder''$ have lower frequency, then the search peer randomly select one peer A in $region($"$wonder''$) and sends a query request to peer A. This query request is routed to A using the CAN routing scheme.

- Step4: ASP Iterative Deepening

  The peer received the query request initials an iterative deepening procedure and returns the search results to searcher.

- Results

1	What a wonderful world
2	What a wonderful car
3	What a wonderful dog

3.8 Adaptive Space Partition

Because keyword query is a zipf-distribution, a few words may have very high frequency, which makes its region contains more than MAX peers. In this case, additional partitions are needed. In the previous example, inverted index entries entries are inserted to $region($"$what''$), $region($"$wonderful''$) and $region($"$world''$). Suppose the there are too many peers in $region($"$what''$), so this region is too big to flood efficiently. It causes no problem because this object can still be easily located by other regions of low frequency keyword. Even queries only contains high frequency keyword can still find lots of relevant objects to that keyword. It still makes sense because users only want to get objects related to the high frequency keyword and they do get them efficiently.

For example, suppose the keyword "$what''$ have a very high frequency, which makes the number of peers in $region($"$what''$) larger than MAX. This means $region($"$what''$) cannot be efficiently searched using flooding message. However, inverted index entries can still be

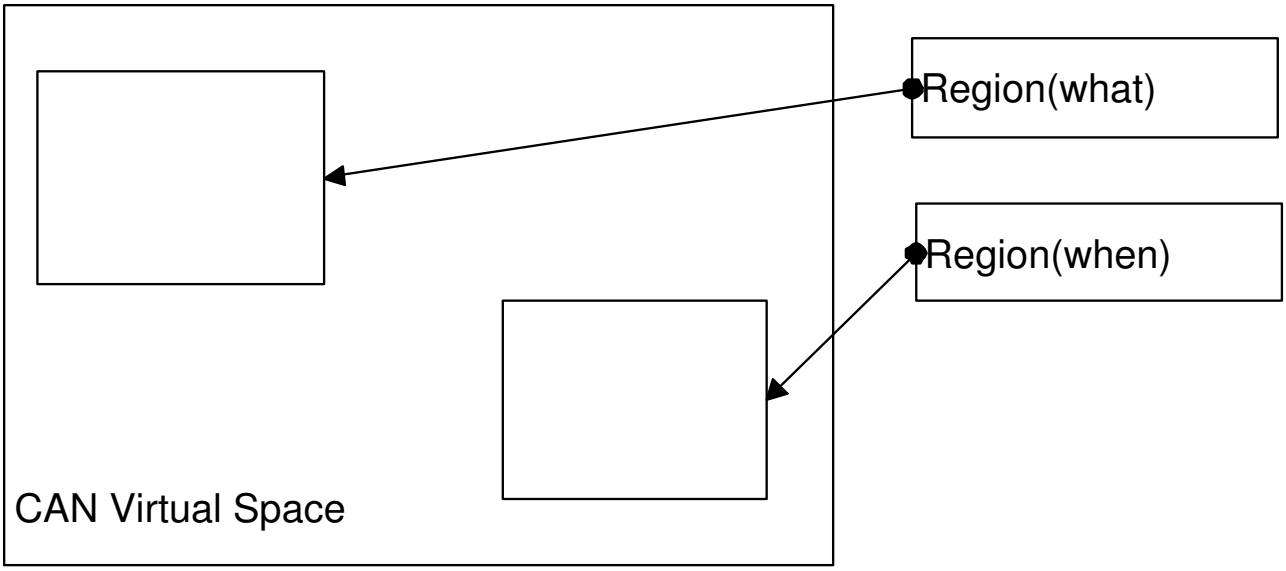

Figure 3.3: object insert

easily found using $region(\text{“}wonderful\text{”})$ or $region(\text{“}world\text{”})$. For the most part, inserting first level inverted index only is good enough.

Problem happens when there are more than two high frequency keywords in the object title. In this case neither of them two (high frequency keywords) can be used to locate this object efficiently. Thus besides the first level inverted index; second level inverted index is also needed. That is what we called *Adaptive Space Partition*. For example, suppose a object, whose title is $\text{“}whatwhen\text{”}$, is inserted.

In this case nether $region(\text{“}what\text{”})$ nor $region(\text{“}when\text{”})$ can be efficiently searched. To solve this problem, the second level inverted index is inserted as fellows.

We randomly select one peer in $region(\text{“}what\text{”}, \text{“}when\text{”})$ and insert entries:

prefix	full title
what,when	what,when

And then randomly select one peer in $region(\text{“}when\text{”}, \text{“}what\text{”})$

prefix	full title
when,what	what,when

The sub-region used the first level region as its base-region and is mapped the same as

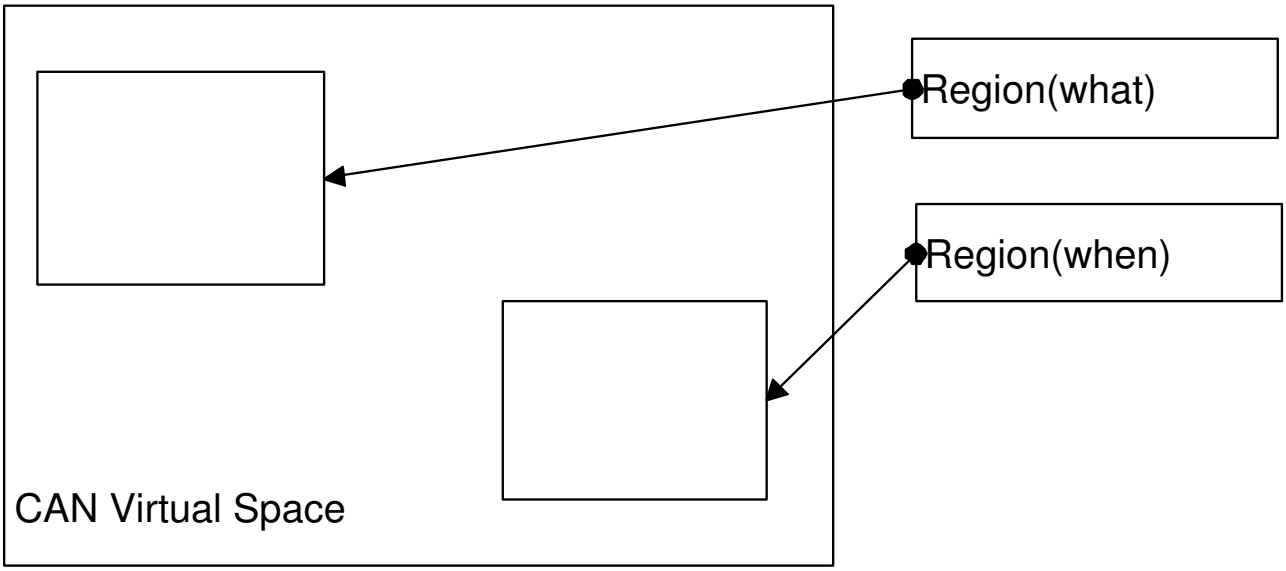

Figure 3.4: need space partition

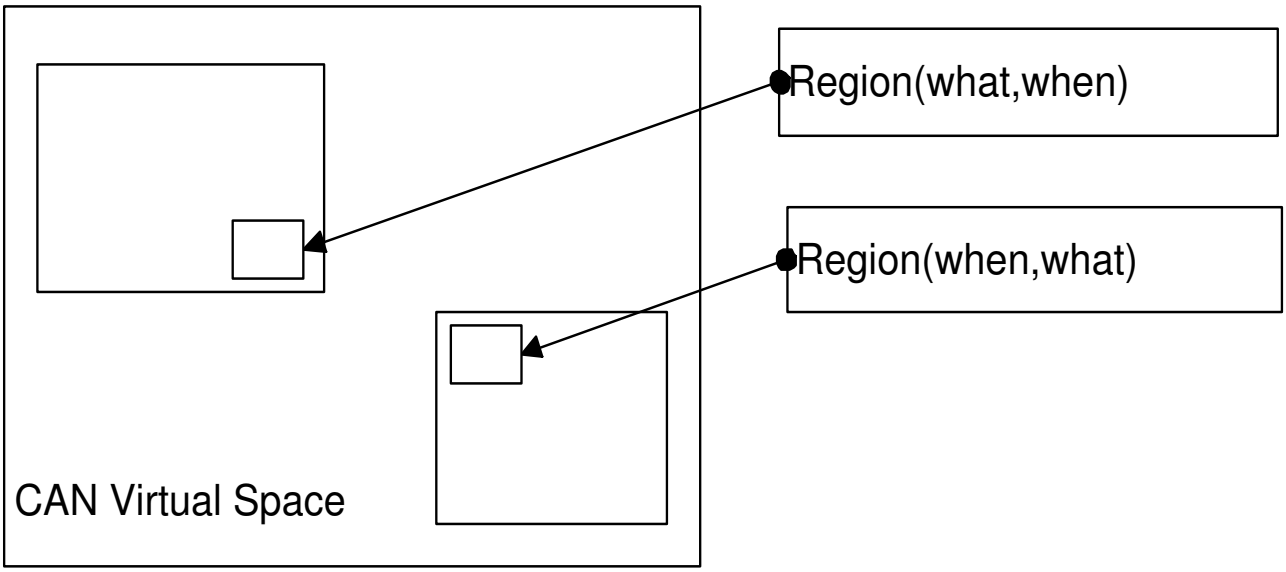

Figure 3.5: after partition

the first level region. If the number of peers in $region(\text{“}what''$, $\text{“}wonderful'')$ is less than MAX, then we are done, otherwise the space partition procedure is recursively invoked.

Because most keywords whose frequency are very high may be consider as meaningless and are ignored during the stemming step. We excepted most partition are finished within two levels.

Note that there's no routing overhead for this adaptive space partition because all calculations can be done locally.

The ASP insert procedure is defined as fellows:

boolean meaningless$(keyword)$
{
return $freq(keyword) > FREQMAX$;
}

boolean toobig$(region)$
{
return $containpeers(region) > MAX$;
}

void addObject$(title)$
{
for (all keyword kw in $title$)
{
if $meaningless(kw)$
$title = title - kw$;
else
$title = title - kw + stem(kw)$;
}
for (all keyword kw in $title$)
$asp(\text{""}, kw, title, canspace)$;
}

```
void asp(prefix, kw , title, baseregion)
{
    region = map (kw , baseregion);
    prefix = append(prefix, main);
    insert(prefix, title, region);
    if (toobig(region))
    {
        title = title − kw;
        for ( each keyword okw in title ) // okw stands for other keyword
        {
            oregion = map(okw, baseregion);
            // for each keywords pair whose keywords are both maped to big region
            if (toobig(oregion))
                asp(prefix, okw, title, region);
                // use region as new baseregion and recursively invoke asp()
        }
    }
}
```

The query procedure is defined as fellows:

```
void query ( query )
{
    region = canspace;
    while ( query is not empty)
    {
        kw = maxfreqkeyword(query);
        query = query − kw;
        prefix = append(prefix, kw);
        region = map(next, region);
```

```
        if ( !toobig(region) )
            break;
    }
    flood(region, prefix, query);
}
```

3.9 Step by Step Example

In this section, we show a step-by-step example to show how the adaptive space partition procedure works in detail. Suppose a ASP system currently has a configuration like that:

PEERS	1000
MAX FREQ	0.04
MAX	10

Now we insert an object, whose title is *a b of c d* , and the keyword frequency is like that:

a	0.01
b	0.02
of	0.05
c	0.03
d	0.04

Because $Freq(\text{“}of\text{''}) = 0.05 > MAXFREQ = 0.04$, in the level-1 space partition, only four inverted index entries are inserted:

Level-1 Inverted Index

prefix	object title	containpeers	action
/a	a b of c d	$1000 * 0.1 = 10$	$10 < MAX$, done!!
/b	a b of c d	$1000 * 0.2 = 20$	need level-2 partition
/c	a b of c d	$1000 * 0.3 = 30$	need level-2 partition
/d	a b of c d	$1000 * 0.4 = 40$	need level-2 partition

In the level-2 space partition, six inverted index entries are inserted:

Level-2 Inverted Index

prefix	object title	containpeers	action
/b/c	a b of c d	$20 * 0.3 = 6$	$6 < MAX$, done!!
/b/d	a b of c d	$20 * 0.4 = 8$	$8 < MAX$, done!!
/c/b	a b of c d	$30 * 0.2 = 6$	$6 < MAX$, done!!
/c/d	a b of c d	$30 * 0.4 = 12$	need level-3 partition
/d/b	a b of c d	$40 * 0.2 = 8$	$8 < MAX$, done!!
/d/c	a b of c d	$40 * 0.3 = 12$	need level-3 partition

In the level-3 space partition, two inverted index entries are inserted:

Level-3 Inverted Index

prefix	object title	containpeers	action
/c/d/b	a b of c d	12 * 0.2 = 2.4	$2.4 < MAX$, done!!
/d/c/b	a b of c d	12 * 0.2 = 2.4	$2.4 < MAX$, done!!

No further space partition is needed, we are done.

3.10 Improved Query Operation

The query can always successfully locate all desired objects if two conditions are satisfied.

- The keyword frequency used in query operation is higher than the one used in insert operation.
- The query string contains enough keywords to partition the CAN space to a region which contains no more than MAX peers.

If condition-1 is not met, query may fail to locate some desired objects available in network. For example, suppose one peer inserts an object: OBJ to network. OBJ contains a keyword: KW so the owner peer uses $OwnerFrequency(KW)$ which is locally

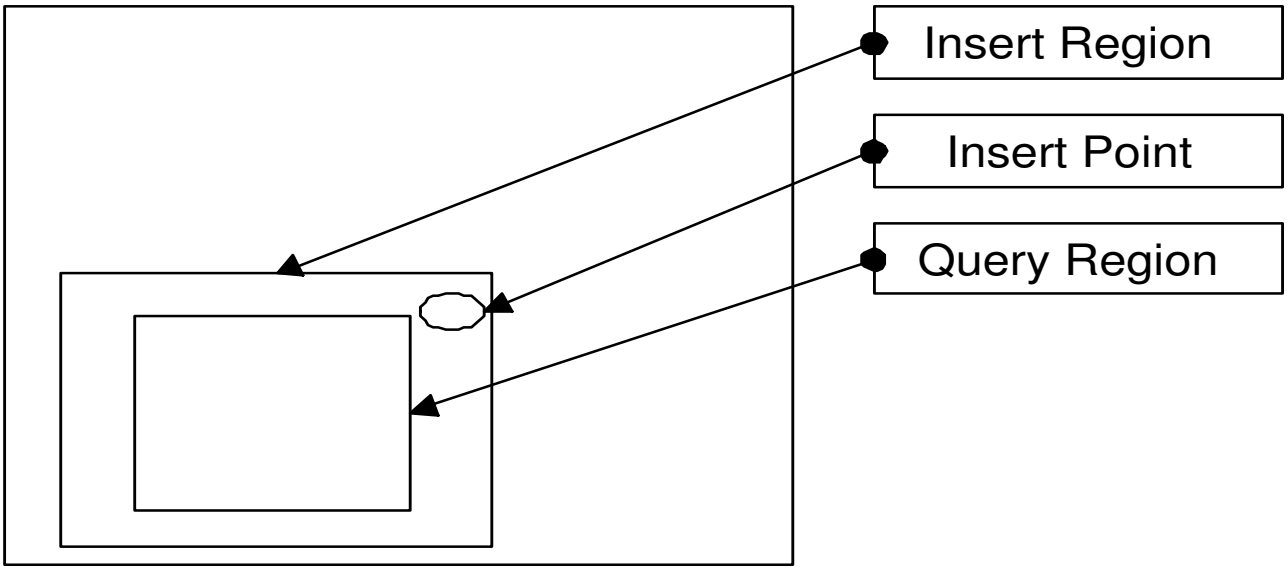

Figure 3.6: Inconsistent keyword frequency

estimated by the owner peer to calculate $region(KW)$. Then the owner randomly selectes an insert point to insert one inverted index entry for keyword: KW. Later, another peer wants to search objects revelant to the keyword: KW. The search uses frequency $QueryFrequency(QFREQ)$ to calculate $region(KW)$ and flood query message in it. Unfortunately $OwnerFrequency(KW)$ is bigger than $QueryFrequency(QFREQ)$ and the insert point falls beyond the region calculated by query peer. As a result, the query operation may fail to locate this object.

To solve this problem, one approach is to make all peers use the same frequency. In this scheme, peers use a global keyword frequency profile, and each peer keeps a copy of this profile. This profile may record only $top-N$ keyword frequency. During insertion and query, peers all use this profile to determine the keyword frequency. Since all frequency estimations are the same, the problem mentioned above does not occur at all. This profile is updated periodically and as we mention above, the keyword distribution in peer-to-peer environment is very stable, thus this profile need only very low update frequency. This approach works, but it need some global knowledge, and may be unacceptable in some situations. Another way is to use a flooding scale factor, this approach cannot eliminate all failure possibility, but it increases the possibility to success. It utilizes the flexibility of flooding. It works as fellows; during a query, the area of the target region and MAX is multiple to a flooding

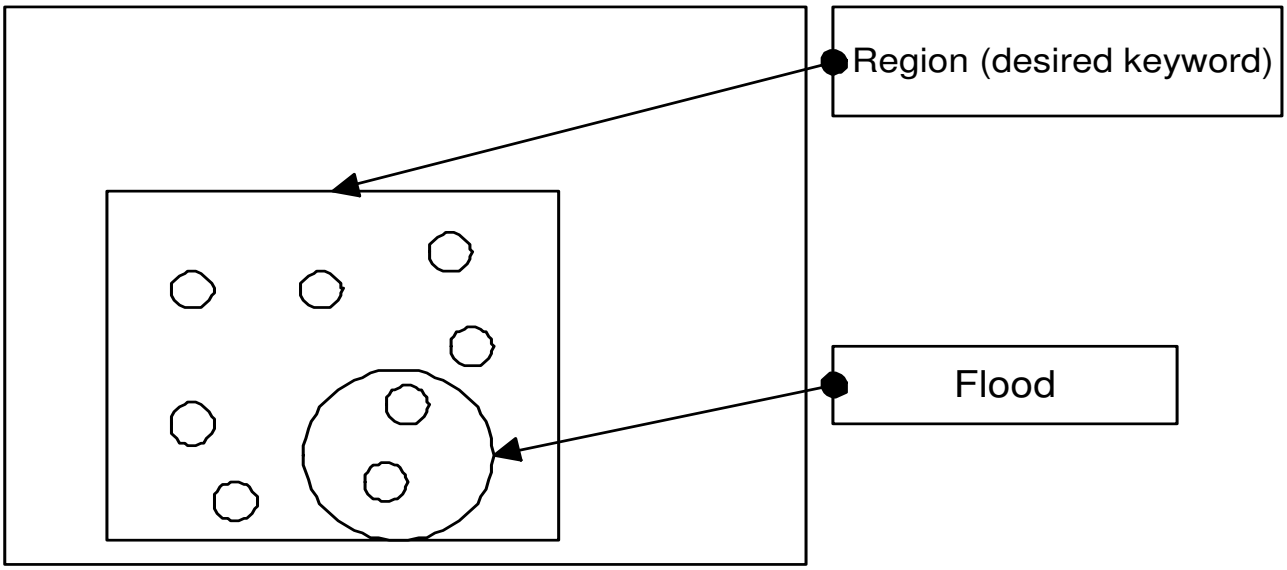

Figure 3.7: Partial Covered Region

scale factor. Scale factor makes the query region bigger and is more likely to cover the insert point. Now we discuss condition-2. If condition 2 is not met, query message cannot flood to the whole target region because of the MAX limit. This causes no problem in most cases. Because objects contains this desired keyword is equally distribution in the region. The expected number of found objects is the same wherever the part to flood.

Problem occurs when the inverted index density is very low, and the query peer wants to find large number of objects, thus query peer may fail to locate enough objects using only one query operation. In this case, all the query peer needs to do is simply invoking the query operation more times. Because different query operation floods query message to different portion of the region and thus get more desired objects, peer keeps on invoking query operation util it found enough objects.

3.11 Persistence Object and Network Expansion

When the number of peers increases, the density of peers increases too. This makes some keywords' region become too big to flood. In most cases, it makes no problem because in pratical peer-to-peer environments, object's life is much shorter than network, the network size is almost stable during an object's life. Though some applications expect objects to

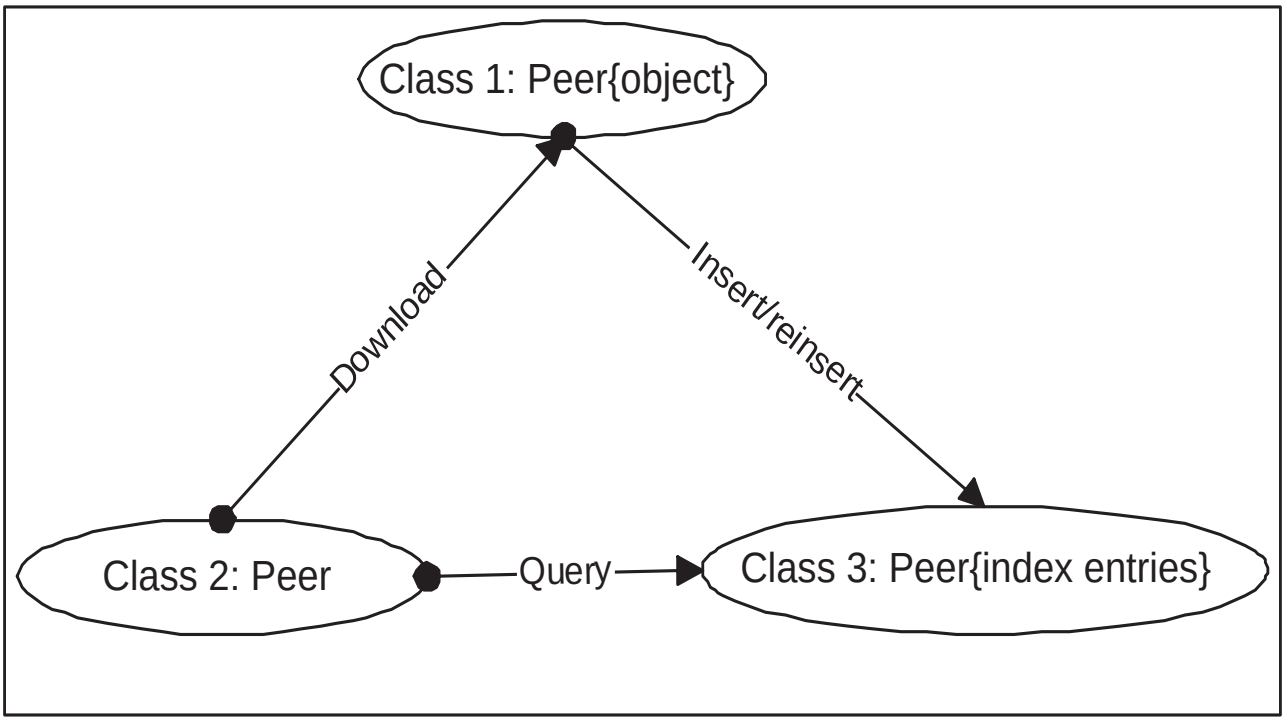

Figure 3.8: Persistent Objects

be persistent. In this case, further space partitions are needed. The problem is: who is responsible for monitoring the network expansion and insert more inverted index entries when needed.

During an object's life, three classes of peers are involved:

1. Peer who stores the object
2. Peers who store its inverted index entries
3. Peers who want to locate (invoke query procedures) the object

In our scheme, it's class-one peer's responsibility to make the object's inverted index entries consistent. To share an object with others, class-one peer first uses the ASP scheme and inserts some inverted index entries to network. The peer also records the inverted index entries it just inserts in. Then the peer periodically recalculates regions for each inverted index entries. If there are some regions becomes too big because of network expansion, the peer partitions them and inserts addtional inverted index entries.

This monitoring procedure is done periodically and random. It makes no network bandwidth overhead because it is just a local computation. Furthermore, since object's life is much shorter than network, it's good enough to invoke monitoring procedures in low frequency.

Chapter 4

Simulation and Analysis

We write a simulator to build a virtual ASP environment, and use titles of mathematical books in library as data source. Our simulation runs both on batch mode and animation mode so we can see how it works in detail. For each experiement, first we setup system parameters and the data type we want to gather, then we start a simulator batch task, the batch task loads all objects and queries in the data source and uses each of them for insertion and query. During each object insertion and query, the simulator also records datas we want to measure.

The simulator exports a simulation report when the batch task finished. Finally we use analysis tools to analyse those simulation reports and generate statistical disgrams.

4.1 Data Source

Our data source is extracted from information systems in Taiwan University Library and contains about 20,000 book records and 19,000 query logs. We use it as data source because it's a very typical zipf-distribution, just like files shared in peer-to-peer environment.

Some object examples are listed below:

Object Example
Elements of functional analysis
Second order partial differential equations of mixed type
Partial differential equations an introduction
Schaum's outline of theory and problems of linear algebra
Logic programming proceedings of the 1991 international symposium

Some query examples are listed below:

Query Example
stochastic integrators equations
Hamiltonian
systems intelligent
Meta-level
Differential

4.1.1 Keyword Distribution

The average length of book titles is 7.421 words. Keyword distribution in book titles is listed below.

The average length of queries is 1.766 words. Keyword distribution in queris is listed below.

As we can see from those figures, both objects and queries are both zip-distributions.

4.2 System Parameter

The system parameter is defined as fellows:

- PEERS: Number of peers
- MAX: Max number of flooded peers
- HASH: Hash function which maps a keyword to a value between 0 and 1. Its value is ether "MD" or "SHA"
- MAX FREQ: Max frequency of meaningful keyword; Keyword whose frequency is larger than $MAXFREQ$ is regarded as meaningless and thus is ignored.

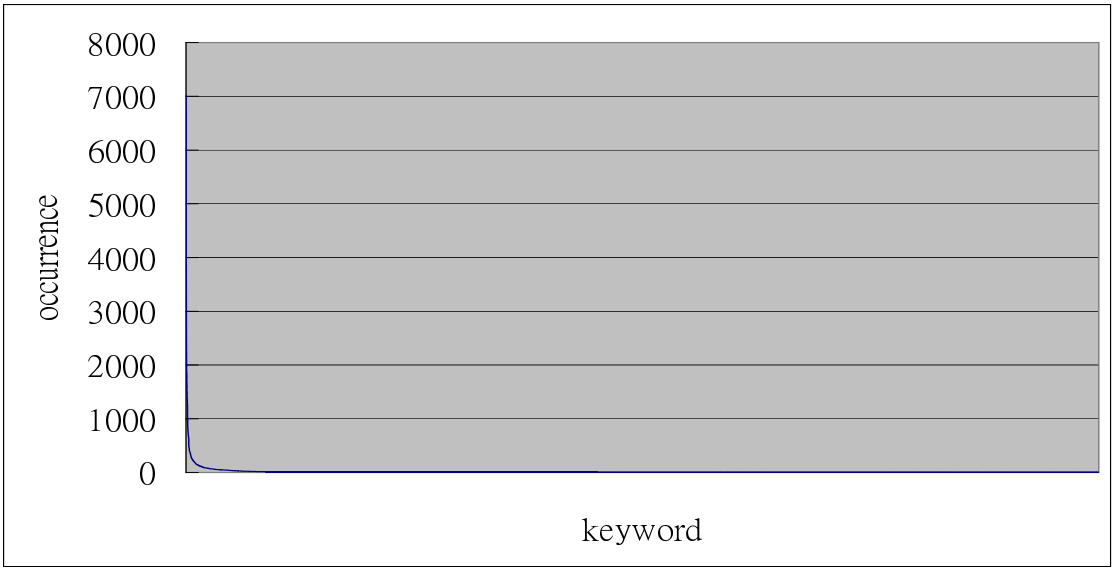

Figure 4.1: Object Keyword Distribution

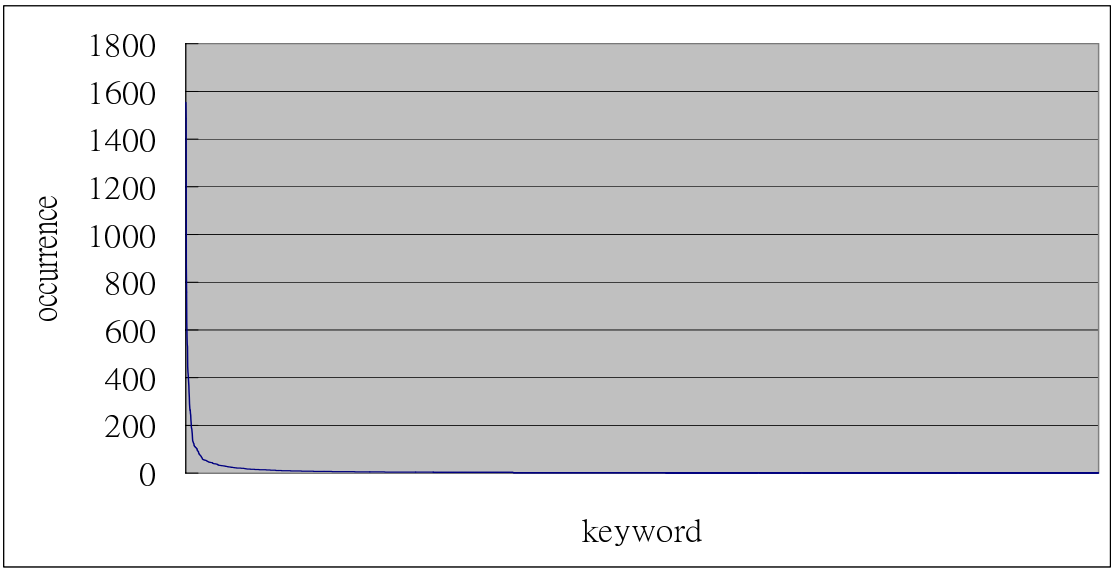

Figure 4.2: Query Keyword Distribution

- FLOOD SCALE: the flood scale factor, which is used to increase the probability of successful search.
- USER WANTED: The average number of objects a searcher wants to find during one search operation.

4.3 Insertion Overhead

The number of inserted inverted index entries determines the main overhead of object insertion. To analyze the scalability of object insertion, we first build the CAN overlay network on different sizes, and then run a simulator batch task. The simulator batch task parse the data source file and insert each object in network. Simulator also recorded the total number of inverted index entries. System parameter and simulation results are listed bellow:

PEERS	10000, 20000 ... 100000
MAX	50 100 150
HASH	"MD5"
MAX FREQ	0.02

As we see in the figure, the number of total inverted index entries is sub-linear and grows slower than the number of peers. Furthermore, when the MAX paremeter is increased, the number of inverted index entries decreases. Because big MAX means big flooding area, which makes the adaptive space partition procedures stop earlier. Note there are about $20,000$ objects records and the average title length is 7.421. In the worst case ($MAX = 50, PEERS = 100000$), the total number of inverted index entries is about $500,000$. and each object insertion need about 25 inverted index entries and each keyword need about 3.3 inverted index entries on average.

In pratical peer-to-peer sytems, the number of total objects grows linearly with the number of peers. If the total number of shared objects can be measured as $O(PEERS)$, the number of objects maintained by a peer is roughly a constant.

$$OnePeerLoad = \frac{O(PEERS)}{PEERS} = C$$

In our experiment, the number of objects is static. So we use the total number of inverted

index entries to measure the insert overhead instead of the average number of inverted index entries for one peer.

The experimental results are listed below:

number of peers	max = 50	max = 100	max = 150
10000	174891	130077	120431
20000	225055	174891	143951
30000	276503	196609	174891
40000	319569	225055	194887
50000	342335	245815	207335
60000	376247	276503	225055
70000	405105	299913	237669
80000	429917	319569	254635
90000	465285	328447	276503
100000	489283	342335	290077

4.4 Query Overhead

A query operation contains two parts. The first one is a CAN routing procedure. The second is our ASP iterative deepening. The overhead of the first one is determined by underplaying CAN and CAN's system parameter so we did not discuss it. We focus on the number of flooded peers for each query.

First we build a $30,000$ peers network. Then we analyze the scalability of the ASP interactive deepening. The simulator parses the query logs and use those query strings to invoke the ASP query operation. Our simulator also calculates the average number of flooding peers for a query. We run this simulation on different $USERWANTED$ to see the overhead of the ASP interactive flooding. System parameter and simulation results are listed bellow:

PEERS	30000
MAX	50, 100, 150
HASH	"MD5"
MAX FREQ	0.02
USER WANTED	5 10 15 20 25 30 35 40 45

Note the average number grows with $USERWANTED$ sub-linearly and is much less

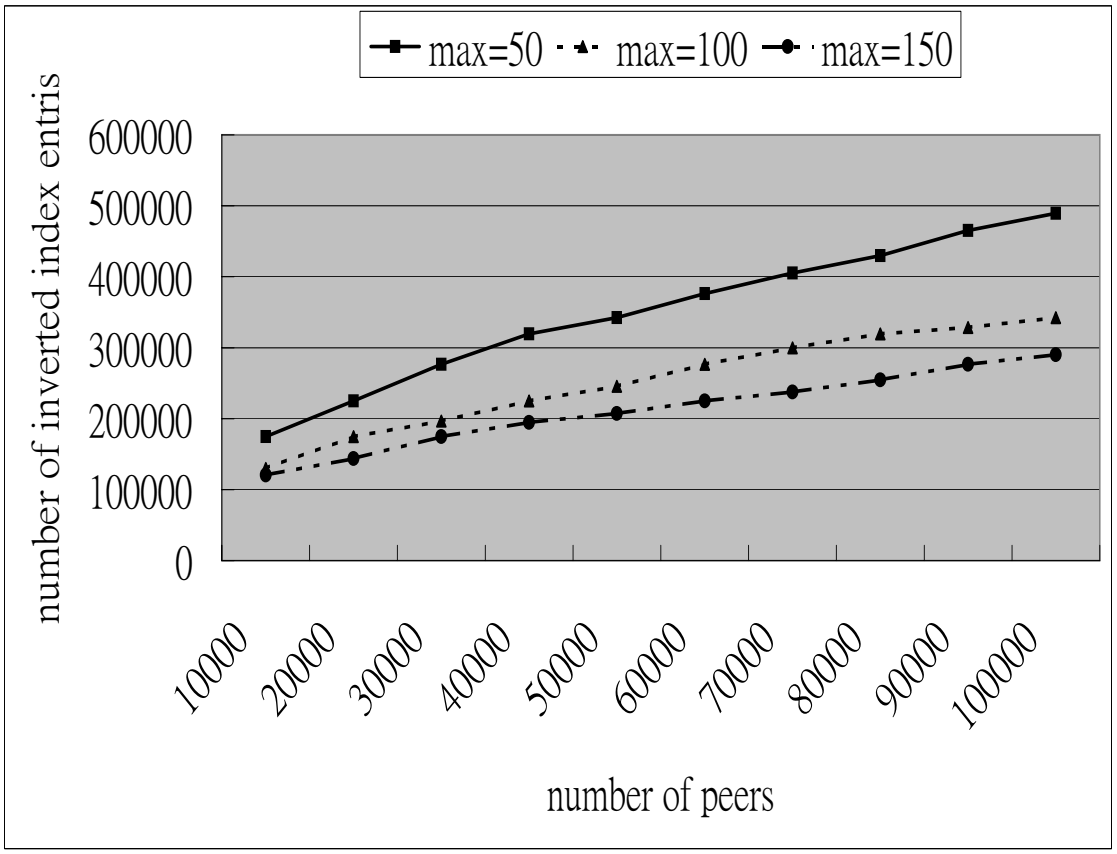

Figure 4.3: Number of Inverted Index

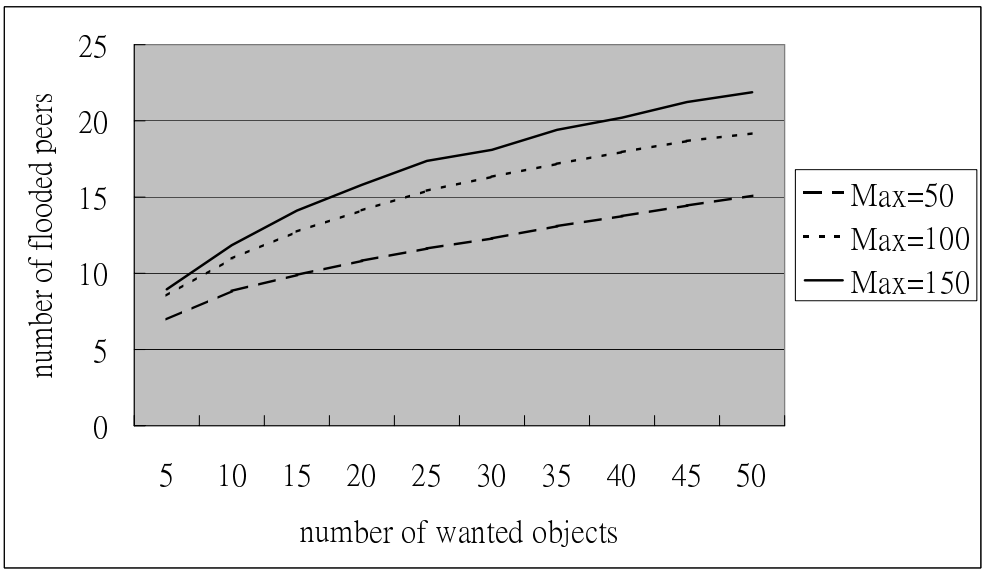

Figure 4.4: Number of Average Flooded Peers

than MAX. Because in the ASP iterative deepening, three conditions stops flooding; keyword region boundary, $USERWANTED$ and MAX. Even in the worst case, no more than MAX peers are flooded in a single query operation. When the number of peers grows up, the ASP iterative deepening would not run out of network bandwidth

The experimental results are listed below:

number of flooded peers	peers = 50	peers = 100	peers = 150
5	6.98	8.54	8.95
10	8.84	10.97	11.84
15	9.89	12.74	14.11
20	10.82	14.13	15.8
25	11.63	15.42	17.37
30	12.29	16.34	18.1
35	13.09	17.18	19.41
40	13.75	17.96	20.21
45	14.44	18.68	21.24
50	15.09	19.18	21.88

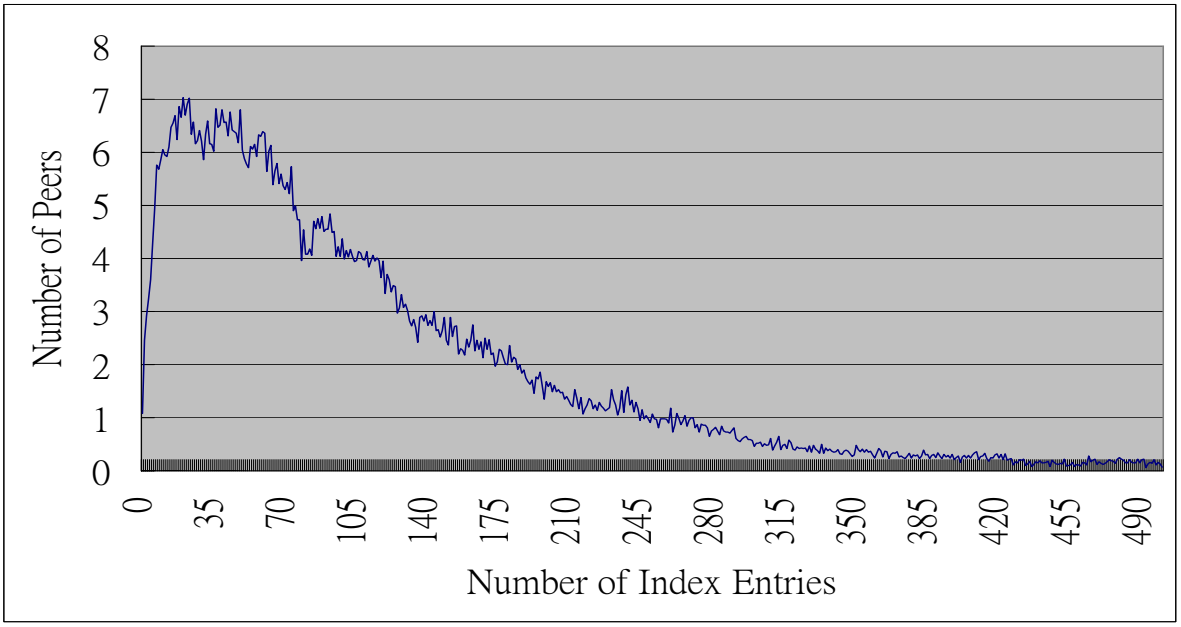

Figure 4.5: Peer Loading Distribution

4.5 Peer Loading Distribution

In this experiment, we analyse load balance of the ASP scheme.

PEERS	1000
MAX	100
HASH	“MD5“
MAX FREQ	0.02

To obtain a smooth distribution curve, we use the system parameters listed above to run the simuation 100 times and calculate the average loading distribution. The number of inverted index entries is 120285. The x-axis values are number of inverted index entries. The y-axis value means how many peers stores this particular number of inverted index entries.

As we see in the figure, the peer loading distribution has a long converge curve. This is because of the zone distribution in CAN. Everytime a new peer enters the CAN network, one peer zone is split. This scheme makes the peer zone area a non-continous distribution.

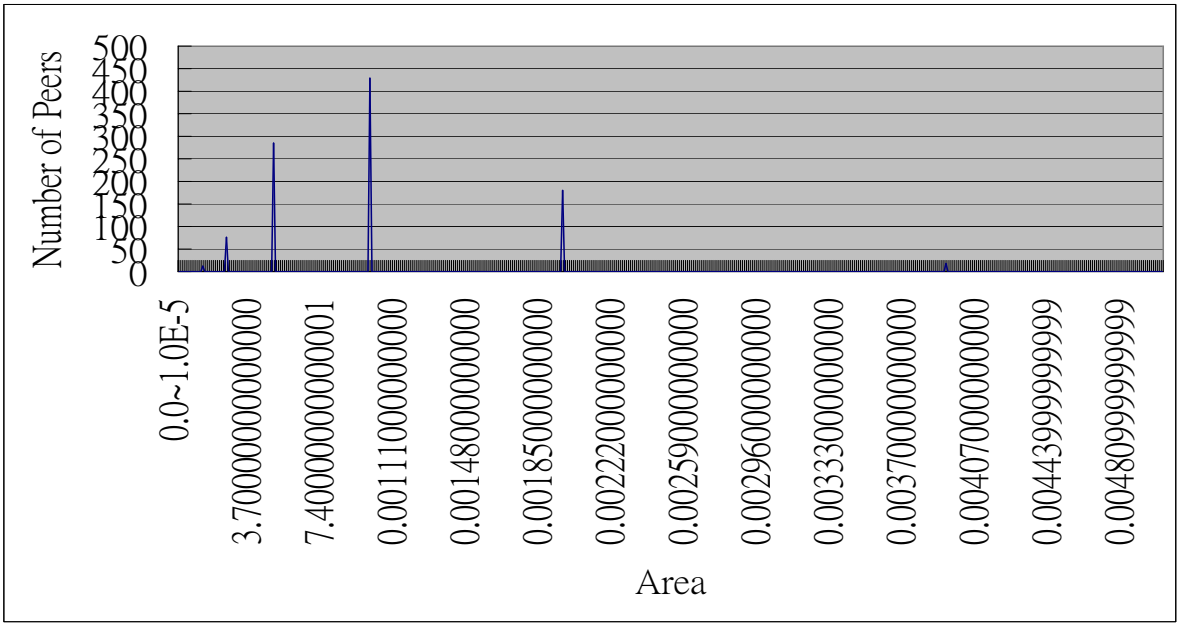

Figure 4.6: Peer Zone Area Distribution

Peer zones falls in separate groups. Peer zones in the same group have the same area. Peer zones in one group have a 2^n multiple area than those in other groups.

This causes no load balancing problem. Because a new peer's insert point is randomly selected and thus distributed evenly in the CAN virtual space. A peer with a larger zone has a higher propability to split in the future. As the time goes by, peers in larger zone group (*larger means larger than average*) have a higher probability to shift to smaller zone group, and peers in smaller zone group have a higher probability to shift to larger zone group.

Since our ASP scheme evenly distribute inverted index entries to the CAN virtual space. Peers with larger zone have more inverted index entries, and the number of inverted index entries stores in a peer is propotional to the peer's zone area.

In this example, if we separate peers in different groups, we obtain six major peer groups, which are named as Group1, Group2 ... Group6, from left to right respectively. Their load

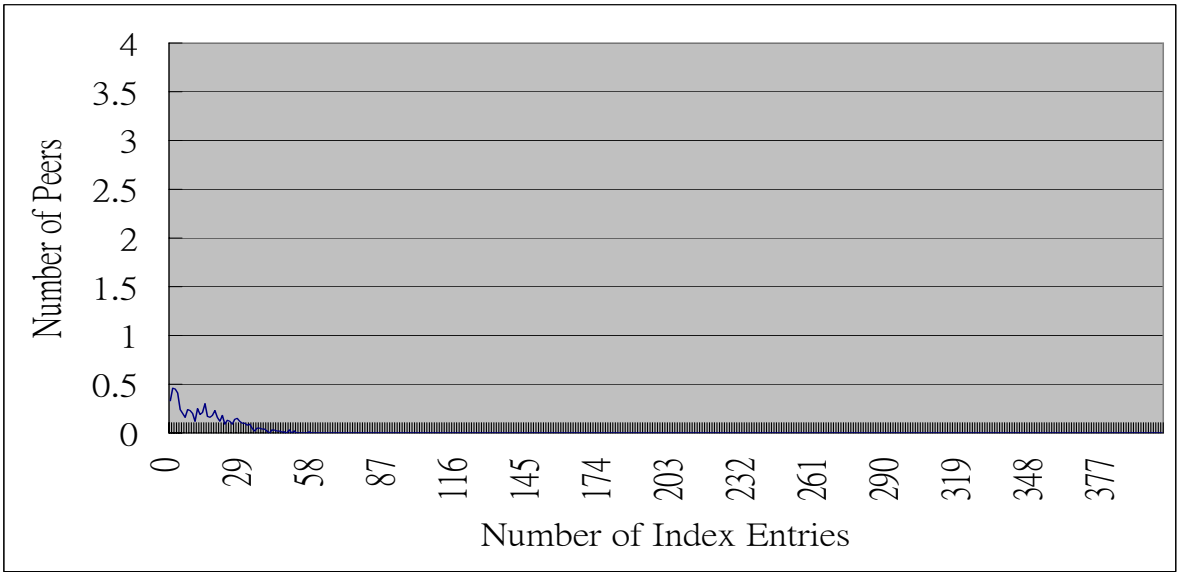

Figure 4.7: Peer Loading Distribution (Group 1)

distribution figures are listed below. As we can see in these figures, peers in larger zone do have more inverted index entries.

Like CAN, this causes no load balancing problem. Because peers who stores more inverted index entries have higher propability to split their zone and handoff their inverted index entries in the future, and the future handoff probability is also propotional to the number of inverted index entries (or zone area).

$$ZoneArea(Peer) \propto NumberOfIndexEntries(Peer) \propto HandoffProbability(Peer)$$

4.6 Inverted Index Level Distribution

Now we analyze the level distribution of inverted index. System parameter and simulation results are listed bellow:

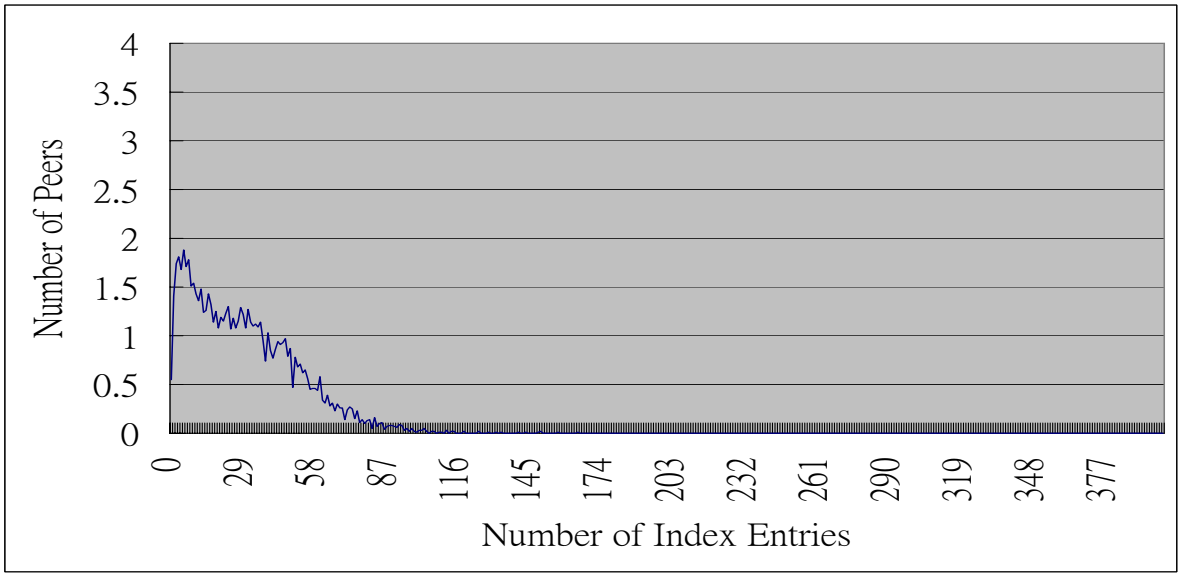

Figure 4.8: Peer Loading Distribution (Group 2)

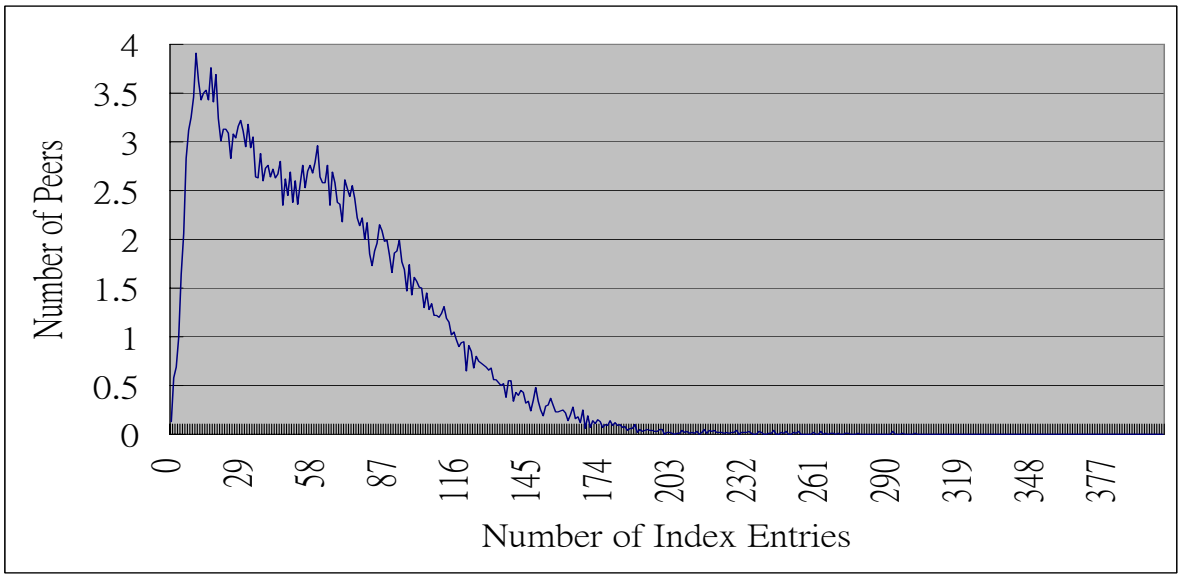

Figure 4.9: Peer Loading Distribution (Group 3)

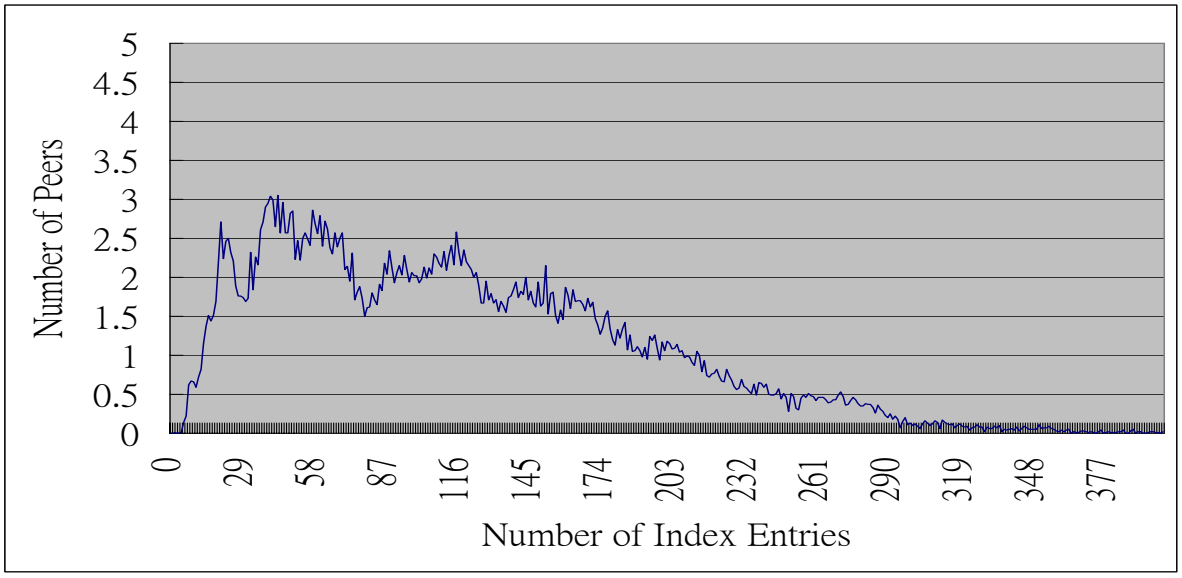

Figure 4.10: Peer Loading Distribution (Group 4)

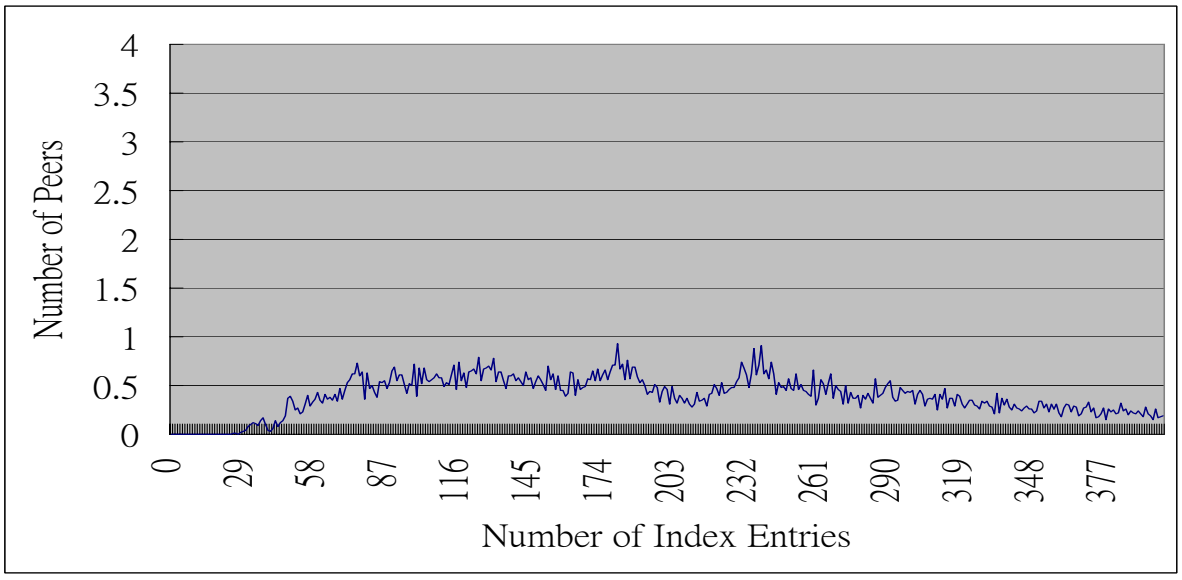

Figure 4.11: Peer Loading Distribution (Group 5)

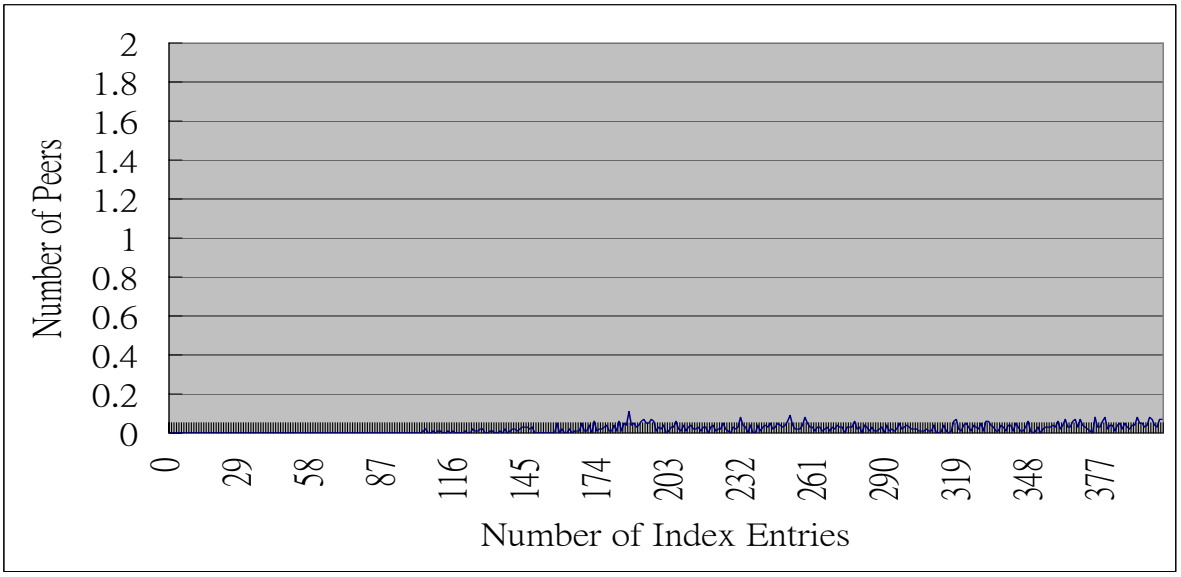

Figure 4.12: Peer Loading Distribution (Group 6)

PEERS	1000 10000 100000
MAX	50 100
HASH	"MD5"
MAX FREQ	0.02

Note there is almost no inverted index entry whose level is bigger than 3. This result is the same as our expectation. Suppose the $MAXFREQ$ is 0.02 and MAX is 100, no level-3 inverted index entry exists until

$$PEERS > 25,0000 = \frac{1}{0.02} * \frac{1}{0.02} * 100$$

We can also use a Level Threshold to bound the number of inverted index entries, thus even if there is a very large number of peers in network and MAX are very small, the level of inverted index doesn't grows infinite. The threshold causes little problem because current research shows most users use only short query string, and most web search engines have a average query length between 1.5 and 2.5. Furthermore, most users tend to use simple query which contains keywords only instead of complex logic control operators. These facts

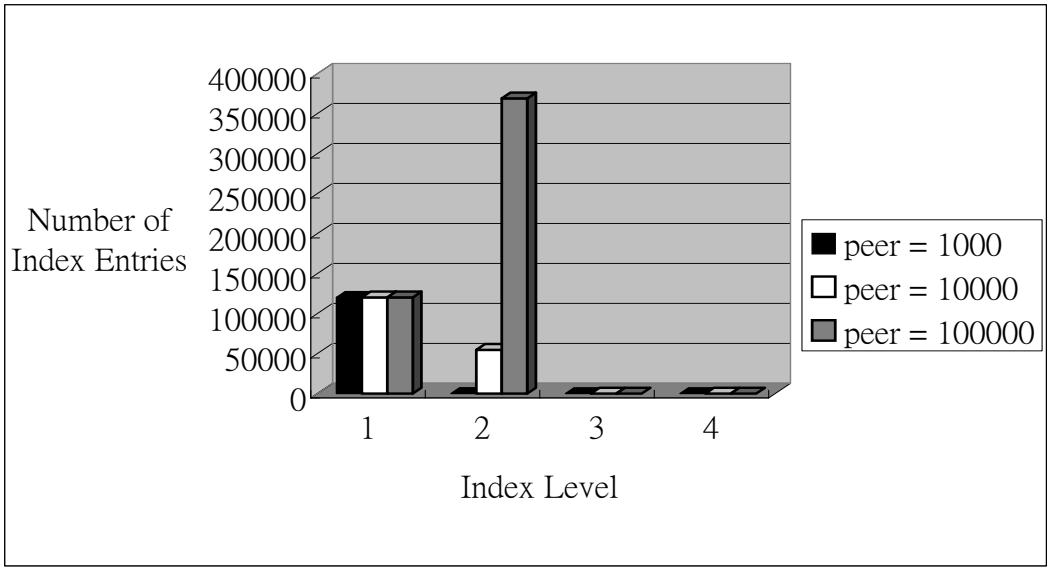

Figure 4.13: Inverted Index Level Distribution, MAX = 50

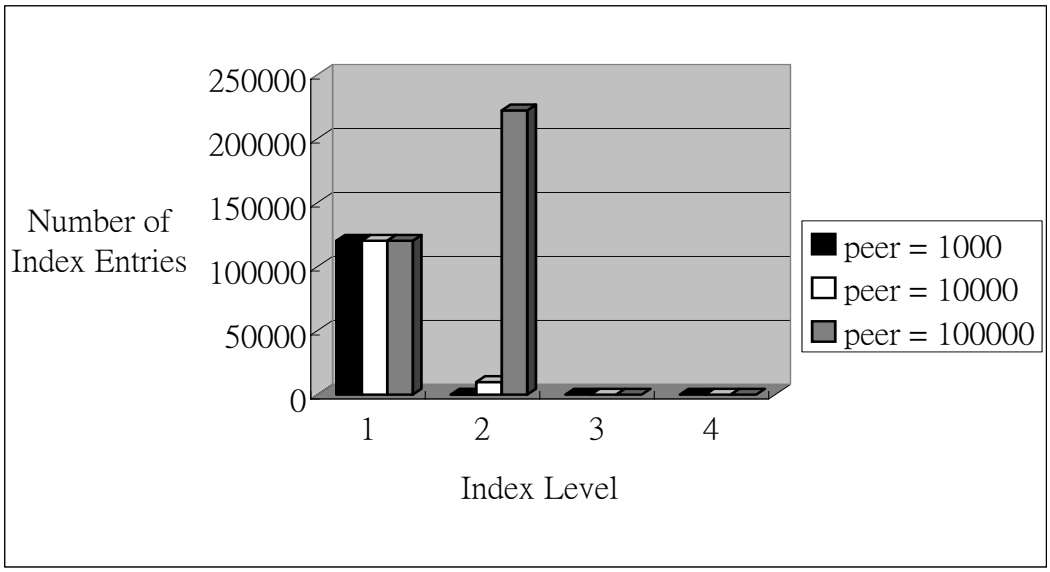

Figure 4.14: Inverted Index Level Distribution, MAX = 100

means inverted index entries whose level is bigger than five are much less useful because user's query string is just not so long.

The experimental results are listed below:

max = 50	peer=1000	peer=10000	peer=100000
1	120239	120239	120239
2	46	54652	369044
3	0	0	0
4	0	0	0

max=100	peer=1000	peer=10000	peer=100000
1	120239	120239	120239
2	46	9838	222096
3	0	0	0
4	0	0	0

Chapter 5

Conclusion

Peer-to-peer applications have greatly broadened the possibility of resource sharing over the Internet. With peer-to-peer applications, resource such as commercial software or unpublished e.books, which were unavailable in the past can now be accessed easily. There are two classes of peer-to-peer networks currently in use: (1) Unstructured peer-to-peer overlay networks are simple, robust, and flexible, but can not scale well. (2) Distributed hash tables were designed to solve the scalability and reliability problems. However, the useful keyword searching features present in Napster and Gnutella are absent from DHTs.

In this thesis, we proposed a hybrid scheme, build on top of the CAN distributed hash tables. Our scheme keeps the flexibility in a unstructured network and still scale as well as a structured overlay network.

Below we summarize the contributions and futute works of this thesis.

5.1 Contributions

In this thesis, we proposed the ASP scheme, which is a hybrid keyword search system that combines a structured and an unstructured peer-to-peer overlay networks. The ASP scheme uses the CAN DHT for message routing and iterative deepening for keyword search. We have two major contributions:

- The ASP scheme has several good properties: first of all, it is load balanced. Because index entries releated to a keyword are distributed according to the keyword frequency. The ASP scheme avoids the keyword zipf-distribution problem. Second, the ASP

scheme is scalable and configurable. By properly setting the system parameters, the ASP scheme can be tuned to suit different environments. Third, the ASP scheme is a hybrid system, which has the robustness of a flooding based system but still as scalable as a DHT. The progressive flooding further reduces the bandwidth requirement, and thus makes lower network load than typical flooding systems. Finally, the ASP scheme trades the insert overhead to optimize the query operation. Since query is the most frequent operation, trade-off between object insert cost and query cost does improve system performance.

- To evaluate the performance of the ASP scheme, we simulated a ASP network in different environments and configurations. We evaluated the object insertion overhead, query overhead, peer load distribution, and index level distribution. We showed that both insertion overhead and query overhead grow sub-linearly, and the complexity of query is optimized to be $O(n^{\frac{1}{d}})$, which is the same as a CAN routing procedure. We also showed that the ASP scheme has balanced loads. The simulation results show that the ASP scheme does provide an efficient, scalable, and flexible keyword search functionality for the peer-to-peer networks.

5.2 Furture Work

Despite our effort to make the ASP scheme a pratical peer-to-peer system, it is still just a prototype in laboratory. Future developments are needed.

- The Keyword Frequency Estimation can be further analysed and improved. The most important one is . Though we proposed two schemes in this thesis, their both have their limitations and disavantages. It may be interesting to know each scheme's characteristics such as scalability, overhead, and estimation correctness.

- In our simulations, we used mathmatical book titles as the data source, and the number of data entries are only 20,000. It may be interesting to gather more data, especially the data in real peer-to-peer networks, to run our ASP scheme, and observe how the ASP scheme works in different keyword distribution.

- We showed that the ASP scheme has the desired properties such as scalability, load balancing, and flexibility. But, we did not compare the ASP scheme with others approachs. Comparisions are needed to show the advantages, disavantages, and the applicable environments of the ASP scheme.

- A prototype of the ASP system was built, which now runs on a simulator only. It is desirable to write a complete, fully-functional application to gather real data on a peer-to-peer overlay network.

Bibliography

[1] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings of Designing Privacy Enhancing Technologies, International Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, CA, USA, July 2000.

[2] http://www.emule-project.net/.

[3] http://www.edonkey.com/.

[4] C. Greg Plaxton, Rajmohan Rajaraman, and Andrea W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, 1997.

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

[6] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. In *Proceedings of ACM SIGCOMM*, pages 161–172, 2001.

[7] Ion Stoica, Robert Morris, David Karger, and M. Francs Kaashoekand Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 149–160, San Diego, California, United States, 2001. ACM Press.

[8] Zipf curves and website popularity. http://www.useit.com/alertbox/zipf.html.

[9] Karl Aberer. P-grid: A self-organizing access structure for p2p information systems. In *Proceedings of 9th Eleventh International Conference on Cooperative Information Systems (CoopIS)*, pages 179–194, Trento, Italy, Septemper 2001.

[10] http://gnutella.wego.com/.

[11] Y. Xie and D. O'Hallaron. Locality in search engine queries and its implications for caching. In *Proceedings of IEEE INFOCOM*, New York, USA, June 2002.

[12] Kunwadee Sripanidkulchai. The popularity of Gnutella queries and its implications on scalability, February 2001.

[13] Sunil Patro and Y. Charlie Hu. Transparent query caching in peer-to-peer overlay networks. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 32–, Nice, France, April 2003.

[14] Dimitrios Tsoumakos and Nick Roussopoulos. Adaptive probabilistic search (aps) for peer-to-peer networks. In *Proceedings of 3rd International Conference on Peer-to-Peer Computing (P2P)*, pages 102–109. IEEE Computer Society, 2003.

[15] Sam Joseph. Neurogrid: Semantically routing queries in peer-to-peer networks. In *Revised Papers from the NETWORKING 2002 Workshops on Web Engineering and Peer-to-Peer Computing*, pages 202–214. Springer-Verlag, 2002.

[16] http://www.napster.com/.

[17] Patrick Reynolds and Amin Vahdat. Efficient peer-to-peer keyword searching. In *Proceedings of ACM/IFIP/USENIX International Middleware Conference*, pages 21–40, Rio de Janeiro, Brazil, Junuary 2003.

[18] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[19] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*, pages 190–201. ACM, November 2000.

[20] Baruch Awerbuch nad Christian Scheideler. Peer-to-peer systems for prefix search. In *Proceedings of ACM Symposium on Principles of Distributed Computing (PODC)*, pages 123–132, Boston, Massachusetts, July 2003.

[21] X. Li and C. Plaxton. On name resolution in peer-to-peer networks. In *Proceedings of the 2nd Workshop on Principles of Mobile Computing*, pages 82–89, Toulouse, France, October 2002.

[22] James Aspnes and Gauri Shah. Skip graphs. In *Proceedings of Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 384–393, Baltimore, MD, USA, January 2003.

[23] Omprakash D. Gnawali. A keyword-set search system for peer-to-peer networks. Master's thesis, Massachusetts Institute of Technology, May 2002.

[24] Chunqiang Tang, Zhichen Xu, and Sandhya Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *Proceedings of ACM SIGCOM*, pages 175–186, Karlsruhe, Germany, August 2003.

[25] M.Berry, Z.Drmac, and E. Jessup. Matrices, vector spaces and information retrieval. *SIAM Review*, 41(2):335–362, 1999.

[26] H. Morikawa K. Nakauchi, Y. Ishikawa and T. Aoyama. Peer-to-peer keyword search using keyword relationship. In *Proceedings of 3rd International Workshop on Global and Peer-to-Peer Computing on Large Scale Distributed Systems (GP2PC)*, pages 359–366, Tokyo, May 2003.

[27] Hector Garcia-Molina Beverly Yang. Improving search in peer-to-peer systems. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS)*, pages 5–14, Vienna, Austria, July 2002.

[28] M.F. Porter. Porter stemming algorithm: An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.