UNIVERSITY OF CALIFORNIA
SANTA CRUZ

PEER-TO-PEER BANDWIDTH EFFICIENT KEYWORD SEARCH FILE STORAGE SYSTEM

A thesis submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Justin S. Lee

December 2018

The Dissertation of Justin S. Lee
is approved:

Professor Chen Qian, Chair

Professor J.J. Garcia-Luna-Aceves

Professor Brad Smith

Lori Kletzer
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Peer-to-Peer Bandwidth Efficient Keyword Search File Storage System

by

Justin S. Lee

There is a growing number of electronic smart devices, a decentralized peer-to-peer file storage system can innovate the way we detach from centralized file storage. The objective of developing a completely decentralized peer-to-peer network is the significant performance, scalability, cost, and reliability advantages over centralized single site systems. In this paper, we propose a completely decentralized peer-to-peer network model that utilizes state of the art set membership data structures to achieve a bandwidth efficient multi-keyword search. A multi-keyword search allows users to quickly find desired files without directly addressing the files. The network model extends consistent hashing to replicate files quickly across the network allowing for a swift bandwidth efficient multi-keyword search. Our file system is called the Peer-to-Peer Bandwidth Efficient Keyword Search File Storage System (BEKSFSS).

This peer-to-peer distributed file system combines a Distributed Hash table with cuckoo filters optimized for high throughput, low latency, and accurate search results. We propose an improvement on an existing P2P file system that utilizes

consistent hashing to replicate files quickly across the network allowing economical multi-keyword searches. Files within the system are replicated across the large P2P network which allows for high availability without bottlenecks of centralized data storage. This infrastructure considers data reliability, availability, and storage overhead through highly distributed file storage.

Chapter 1

Introduction

Peer-to-peer (P2P) network research is growing quickly for computing and distributed storage solutions. Such solutions must support a file searching without overloading the network. Many P2P file sharing/storing applications provide scalable alternatives to conventional server-based systems but lack a robust searching algorithms and techniques. Unstructured P2P systems that distribute the values pseudo randomly using Distributed Hash Tables (DHT) provide highly scalable solutions with lookups in $O(\log n)$ overlay routing hops for an overly network of n hosts. However, DHT's only support "exact-matches" which is perfectly fine for directly addressing files. Decentralized file storage systems can offer high reliability, scalability, and high throughput compared to single-site systems. In previous works, P2P search algorithms leverage [10] Bloom filters [3] to save network band-

width by filters between nodes rather than file identifiers between peers during multi-keyword searches. There has been significant advances in set membership filters since Reynolds' paper [10]. Cuckoo Filters [5] provide optimal membership deletion which is vital for the accuracy of the search results. Probabilistic filters that offer deletion were complex, high-cost, and large until Cuckoo filters. The principal contribution of this work is a P2P file storage system with a bandwidth efficient searches only using Cuckoo filters that dynamically update upon file insertions and deletions. Our work is layered on top of Chord which allows for a distributed lookup protocol for a single key. Files are distributed throughout the network based on associated keywords. This work combines state-of-the-art probabilistic data structures and reliable peer-to-peer distributed system practices to achieve an optimal search algorithm for our P2P network. As more devices are connected to the net, decentralized P2P storage has significant scalability advantages over a centralized data storage system. If storage space scales dynamically with new devices and data, a completely decentralized file-system would require no additional resources apart from the participating devices within the network.

Chapter 2

Related Works

2.1 Peer-to-peer systems

Peer-to-peer file systems have significant advantages over single and structured file storage systems. When executed properly, an ideal peer-to-peer file system is completely decentralized, scalable, and has an efficient algorithm for searching for specific files quickly. The first generation of P2P systems searches were centralized or required query flooding such as Napster [2] and Gnutella [1] respectively. Napster uses keyword file associations for centralized searches with a well-connected subset of servers which provides simplicity but is not completely decentralized with a single point of failure. Gnutella has an effective search implementation that floods a search query to all reachable nodes and waits for query responses.

Although Gnutella does not rely on a centralized index, search latency and bandwidth cost increase linearly with network size. Gnutella networks with high node churn rates are not stable enough to support such primitive search algorithms. Since every search must communicate with every other node, the flooding algorithm will completely saturate P2P connections with high bandwidth costs as the network scales upwards. To effectively achieve high scalability in a peer-to-peer system, distributed hash tables (DHT) were utilized for routing and locating data. Structured P2P networks such as Tapestry[14], Pastry[11], and Chord[12] guarentee file discovery in $O(\log n)$ network hops for a network of n nodes. Although the previous DHT-based approaches have efficient key lookups, files can only be located through globally unique id's. Users without a unique id will be unable to locate such files without a keyword-based search.

2.1.1 Chord

Chord is a scalable decentralized structured P2P system that uses an overlay topology that precisely locates and stores data objects based a hashing algorithm and a unique key. A chordal network utilizes consistent hashing allowing for $O(\log n)$ lookup messages, $\log n$ routing table space on each node, well-defined locations for each data object, natural load balance, no imposed naming structure, and minimal network disruption on joins leaves. Each node only needs to know

the routing information for $O(\log n)$ other nodes for efficient routing and a lookup has an $O(\log n)$ complexity. In other words, each node maintains a routing table, finger table that is able to route a request at least halfway close to the node responsible for a given data object.

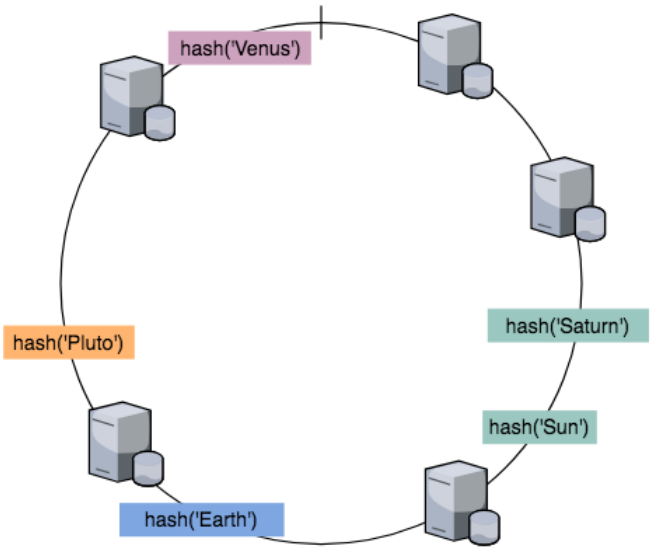

Figure 2.1: Consistent Hashing Ring

As seen in Figure 2.1 the consistent hashing ring network hashes and maps each server's unique identifiers (Ex: IP number) and keys onto the ring. Keys are stored on the next server in the "clock-wise" direction on the ring. Keys and servers are arbitrarily placed on the ring, the network is balanced based on the cryptographic hash.

Consistent hashing [7] is used within Chord to assign keys to Chord nodes. When the number of nodes changes, there is little work required to rebalanced

the network. Rather than having to rehash every key and node identifier when a node leaves and joins a network, the only the nodes "before" and "after" the node on ring redistribute its data. The three steps that must occur when a new node is added are initializing fingers and predecessors, updating fingers of existing nodes, and transferring any keys that should be relocated onto the new node. When a node leaves the network, the node's successor, node in the clockwise or counterclockwise direction on the ring will store all keys stored on the leaving node. The novelty of Chord is to naturally build a completely decentralized network that stores, deletes, and looks up data object in $O(\log n)$, n peers. Peer-to-Peer Bandwidth Efficient Keyword Search File Storage System layers on top of Chord by replicating files across nodes based on the hash of a file's associated keywords. For example, a file named "Harry Potter" would be stored at every node that is is responsible for "Wizard", "Harry", "Potter", and "Scar".

2.2 Probabilistic Filters

Probabilistic filters are space-efficient, data structures that quickly approximate set-membership. These filters can return conclusive negative results with a low probability of false positive results.These filters state when an element is definitely not in the represented set of elements or might be in the represented element set. Filters are used to yield faster negative results when querying for

elements within a set. For BEKSFSS, filters are used to represent the files held on each network node for given keyword. The filters are updated as files are stored and deleted from the node. Probabilistic filters are best used when retrieving data within the represented data set is costly. Probabilistic filters have many network related applications such as resource routing, packet routing, and collaboration in overlay and P2P networks. Bloom filters are the most popular probabilistic filters due to its size efficiency and simplicity. This paper will simulate and calculate the practical benefits of Cuckoo filters over Bloom filters.

2.2.1 Bloom Filters

Bloom filters are compact representations of a set of items and allows for quick access to the membership of a given set. Bloom filters are bit-vectors that approximate set memberships by setting the corresponding locations of data elements within the vector. Its bit-vector is initialized to all 0's with a specified length n with k hash functions. When a new item is inserted into the set, the Bloom filter is updated with the following insert function which is completed in $O(k)$ time.

Insert Technique : Data object's key is hashed by k hash functions to get k positions. For every position that bit is set to '1' in the filter.

Lookup Technique: Data object's key is hashed by k hash functions to get k positions. If all of the positions are set to 1, the data object is possibly a member

of the set. There is no way to distinguish whether the corresponding positions were set by the insertion of another object or the queried object. If not all associated positions within the vector are set to 1, the lookup function will state that the queried data object is not in the represented set.

In Figure 2.2, there is a bloom filter using 2 hash functions. An insert sets at most 2 different bits within the bit vector to 1. A lookup of element e will reveal that e is definitely not in the set. A lookup of element f will reveal that is an example of a false positive.

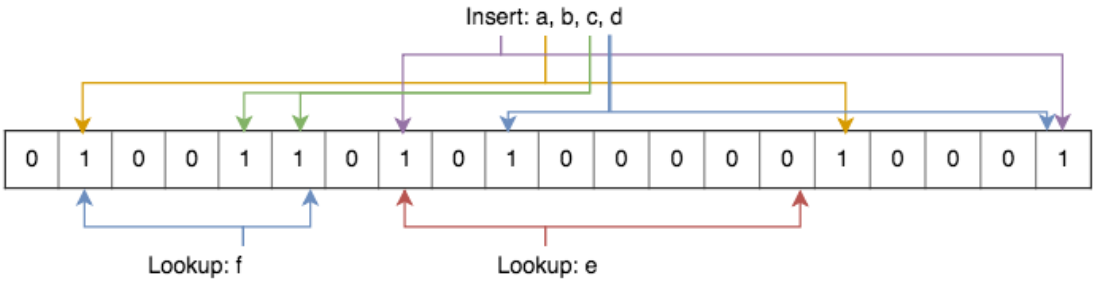

Figure 2.2: Bloom filter

The power of a Bloom filter is determining the set membership of search queries with a low spacial overhead. Bloom filters return results within $O(k)$ time, given the filter uses k hash functions. Bloom Filters have a false positive rate that is dependent on the size of the bloom filter and k hash functions used. The false positive rate is $(1 - e^{-kn/m})k$ where n is the maximum number of elements you'd expect in the set, m is the total number of bits within the Bloom filter or the total

size, and k is the number of hash functions used within the Bloom filter.

Bloom filters only have insert and lookup functions and do not allow the deletion of items within the filter when removed from a data set and must be remade to represent the data. While regenerating filters is simple, recreating the filter is at the cost of filter availability based on set size.

Counting Bloom filters are a modification of bloom filters allowing key deletion which by holding a running counter rather than a single bit for a position. The counter at each position is incremented and decremented upon insertion and deletion from the filter, respectively. The addition of a counter allows for duplicate entries at the cost of requires at least 4 times more space than a standard Bloom filter when each counter consists of 4 or more bits.

2.2.2 Cuckoo Filters

Cuckoo filters, an alternative to Bloom filters represent the approximated set memberships. In contrast to Bloom filters, Cuckoo filters support the insertion, lookup, and deletion of elements from the filter. The filter is a variant of the cuckoo hash table [9] that holds fingerprints, bits strings derived from specific hash functions. Cuckoo filters offer low false positive rates with faster time when compared directly with Bloom filters of the same size [5]. Cuckoo filters have around 95% space utilization which can offer equal false positive rates in compar-

ison with Bloom filters. Both Bloom and Cuckoo filters sizes are determined by desired false positive rates. In practical applications with a target false positive rate, ϵ less than 3, cuckoo filters use less space than a bloom filter.

Cuckoo filters offer four notable advantages over Bloom filters:

1. Dynamic insertion and deletion of elements without rebuilding the entire data structure.

2. Can represent much larger sets than traditional Bloom filters with the same size [5].

3. Constant false positive rates with up to 95% space utilization.

4. Requires less space than a Bloom filter for practical applications that require a false positive rate ϵ is less than 3% [5].

Cuckoo filters consist of an array of buckets that can hold multiple entries that are initially empty. A unique key for each file is reduced to a constant-sized fingerprint via a hash function. The insert algorithm uses two different hash functions to determine two possible positions for a data element's finger prints within the filter. Cuckoo filters propose a technique called partial key cuckoo hashing which determines an alternate location for data based on its original fingerprint which is crucially important because the original data item cannot be accessed from the filter. The Cuckoo filter insert algorithm can be seen below as

Algorithm 1.

Algorithm 1: Insert(x) [5]

```
1  f = fingerprint(x);
2  i_1 = hash(x);
3  i_2 = i_1 ⊕ hash(f);
4  if bucket[i_1] or bucket[i_2] has an empty then
5      add f to that bucket;
6      return Done;
7  // must relocate existing items;
8  i = randomly pick i_1 or i_2;
9  for n = 0; n ¡ MaxNumKicks; n++ do
10     randomly select an entry e from bucket[i];
11     swap f and the fingerprint stored in entry e;
12     i = i ⊕ hash(f);
13     if bucket[i] has an empty entry then
14         add f to bucket[i];
15         return Done;
16 // Hashtable is considered full;
17 return Failure;
```

The lookup algorithm, Algorithm 2 checks two bucket positions for an elements fingerprint and returns true if either of the buckets has the element's fingerprint, f. While Bloom filters return results within $O(k)$ time, Cuckoo filters lookup requires exactly 2 lookups, $O(2)$ time. The key feature that sets Cuckoo filters apart from Bloom filters is the ability to delete fingerprints of keys with less space and as accurate as a counting Bloom filter [5]. In a peer-to-peer file sharing system, files should be available for deletion without the cost of availability and search accuracy.

Algorithm 2: Lookup(x) [5]

```
1 f = fingerprint(x);
2 i_1 = hash(x);
3 i_2 = i_1 ⊕ hash(f);
4 if bucket[i_1] or bucket[i_2] has f then
5     return True;
6 return False;
```

The delete function is very similar to the lookup function but removes the matching fingerprint from the buckets if found. Deletion of items can only occur when items have previously been inserted. An efficient delete function is crucial for a P2P file system that relies on the accuracy of its filters.

Algorithm 3: Delete(x) [5]

```
1 f = fingerprint(x);
2 i_1 = hash(x);
3 i_2 = i_1 ⊕ hash(f);
4 if bucket[i_1] or bucket[i_2] has f then
5     remove a copy of f from this bucket;
6     return True;
7 return False;
```

2.3 Efficient Peer-to-Peer Keyword Searching

As stated earlier, DHT's provide efficient lookups but lack keyword-based search queries. In a DHT-based network, a client must know the exact ID or key to locate a specific file. While it is important to evenly distribute keys across a network with efficient lookup, a Chord network does not allow a client to lookup files without its unique identifier.

In *Efficient Peer-to-Peer Keyword Searching* [10], Patrick Reynolds and Amit Vahdat present a novel peer-to-peer search infrastructure. Their implementation suggests distributing the files by its keywords and utilizing bloom filters to represent data within the file storage system. A file insertion into the their system replicates the file across all nodes responsible for a given keyword. Each node

holds Bloom filters that represent the files stored for a given keyword. They show a scalable infrastructure where message traffic grows sub-linearly as the network size increases linearly. The authors were able to reduce consumed network resources and search latency by using Bloom filters.

Without the use of Bloom filters, an ineffective search technique would be to query each node responsible for a keyword within a user's keyword search and return the intersection of all of the results. They proposed a method for multi-key searches by sending Bloom filters to represent data rather than the data itself across the network. Considering a simple two keyword search that requests files associated with both keywords is shown in Figure 2.3. Server S_a holds all files associated with keyword k_a and server S_b holds all files associated with keyword k_b. In an simple search without utilizing Bloom filters, S_a must find the local files with k_a and send every document IDs to S_b. Then, S_b would determine which local files had matching document IDs in the list and k_b to return a list of document IDs to S_a. Server S_a would relay the files associated with k_a and k_b to the user that searched for k_a and k_b. The bandwidth cost of such search grows linearly with the number of documents within the system.

Their system minimizes the wide-area bandwidth consumed by multi-keyword conjunctive searches and improve to overall performance of individual queries, as shown in the following Figure 2.3. Server S_a sends a Bloom filter $F(k_a)$ to

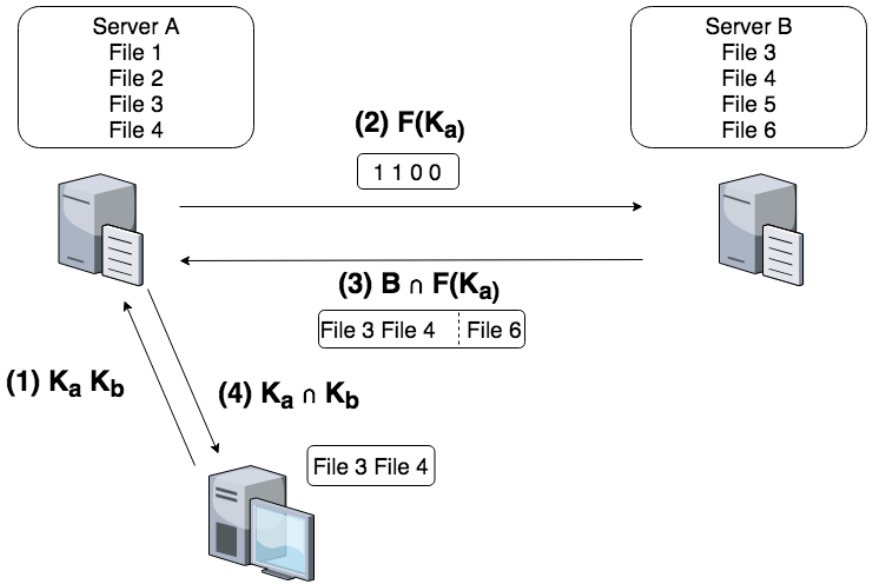

Figure 2.3: Efficient Peer-to-Peer Keyword Search Algorithm

approximate the files associated with k_a, $Files_a$ to server S_b. Server S_b returns the $B \cap F(k_a)$, the intersection between all files on S_b and the Bloom filter for k_a back to S_a which is a list of unique file IDs. Server S_a removes false positives by determining $Files_a \cap (B \cap F(k_a))$, the intersection between the the File IDs from S_b and the files on S_a, $Files_a$. Then S_a returns the files to the client. This search method is able to return the desired search result as $Files_a \cap Files_b$ to the user while only sending one Bloom filters and a short list of File IDs between network servers. The false positives within $F(k_a)$ does not hurt the accuracy of the search because the the false positives are eliminated locally on server S_a right before the final results are returned. The total data sent throughout the exchange

ignoring the results and the initial query are the document ID's of $B \cap F(k_a)$ and the Bloom filter $F(k_a)$. The bits sent from S_a and S_b remains constant even as the number of files within the system grows because of the constant size of a Bloom filter.

This core search technique can be expanded towards any sized multi-keyword search. False positives must be removed which requires the intersections must be sent twice to $n-1$ servers. This means for n keywords, excluding the initial query and the search results, the search requires $2(n-1)+1$ message transmissions. Bandwidth used per search is significantly reduced from a naive search utilizing Bloom filters to approximate the files with keyword_a and keyword_b.

Chapter 3

Peer-to-Peer Bandwidth Efficient Keyword Search File Storage System Model

Our work builds on developing a better file system with a bandwidth efficient search for peer-to-peer networks. The key optimizations proposed in our work is allowing for file deletion, high-throughput searches that leverage Cuckoo filters, and lower multi-keyword search latency. In *Efficient Peer-to-Peer Keyword Searching* [10], file deletion is not discussed. With the addition of file deletion within the probabilistic filters, filters can more accurately represent data within the network and will not require final checks to eliminate false positives. As a

chordal network, nodes can freely join and leave the network. Any node can locate the server responsible for the any specific keyword within the system within $O(\log n)$ network hops. Nodes are added into the consistent hashed ring network by hashing its IP address. This allows natural and dynamic load balancing within the network because of the ability of consistent hashing to allocated keys and nodes evenly especially with virtual nodes [12]. In our consistently hashed file storage system, the keywords associated with a given file are hashed to determine which nodes within the system will hold store a copy of the file. A file with k keywords will be replicated on $\sim k$ different nodes. For the sake of simplicity, we assume that every file that is stored onto the network has at least k keywords for the remainder of this paper.

File storage systems should insert and delete files dynamically without sacrificing availability and latency. Bloom filters cannot delete elements that it represents and must rebuild the entire filter to represent a set of elements. In *Efficient Peer-to-Peer Keyword Searching* [10], searches that require the filter that is being rebuilt must halt until the filter rebuild is completed. A filter rebuild re-adds every file within the node that corresponds with the keyword. Cuckoo filters allow for dynamic file insertion and deletion without rebuilding the entire data structure. The inclusion of Cuckoo filters makes our file system highly available.

We designed the system believing that files searches occur more often than

files are stores and deletions from the file storage system. There are two different types of search options for users, an explicit search and a throughput search. The explicit multi-keyword search guarantees that all files returned are **only** the intersection between all keywords. The throughput multi-keyword search returns all files associated with all keywords within the search with a low probability that a small subset of files are not within the intersection of all keywords.

A file insertion is shown in Algorithm 4 and file deletion is shown in Algorithm 5 below.

Algorithm 4: Insert into file system

Data: Filename, keywords[ ]

Result: Successful file insertion

```
1 for keyword in keywords[ ] do
2     location = hash(keyword);
3     Add file into node @ location;
4     Update Cuckoo filter for keyword @ location;
```

Algorithm 5: Delete file from file system

Data: Filename, keywords[ ]

Result: Successful file deletion

```
1 for keyword in keywords[ ] do
2     location = hash(keyword);
3     Remove file into node @ location;
4     Update Cuckoo filter for keyword @ location;
```

3.0.1 Utilizing the Power of Cuckoo Filters

A flaw in the previous work[10] is not supporting the deletion of a file within the peer to peer network. At the time the paper was written, Bloom filters were the most space efficient set approximation data structure. In a peer-to-peer file storage system using Bloom filters, removing files from the system will always increase false positive rates and waste network bandwidth. For the proposed system to maintain correctness, the Bloom filters must be rebuilt. When a file with k keywords associations is removed from the network, k Bloom filters must be remade. Peer-to-peer networks advertise high availability and keyword searches for any of those k keywords will be unavailable during filter rebuilding times. Even worse, a Bloom filter rebuilding phase can finish and still maintain inconsistency if another file deletion occurs with files with similar keywords.

Cuckoo filters have distinct advantages over Bloom filters for a P2P file storage

system as pictured in Table 3.1. Rather than rebuilding an entire Bloom filter for a single file removal, cuckoo filters can be updated for file insertions and deletions in $O(1)$ time complexity. The differences between Cuckoo filters and standard Bloom filters can be seen in Table 3.1. Counting Bloom filters are not included in the figure because their size in much larger than Cuckoo & Standard Bloom filters for the same false positive rates. g is amount of hash functions used for the Bloom filter. n is the number of files within the network node

	Cuckoo Filter	**Standard Bloom Filter**
Insert	Variable. $O(1)$	Fixed. $O(g)$
Lookup	$O(1)$	$O(g)$
Delete	$O(1)$	Rebuild. $O(n)$

Table 3.1: Cuckoo filter and Bloom filter Comparison

Within BEKSFSS, every keyword has an associated Cuckoo filter that is stored on the node. The Cuckoo filter is updated during file inserts and deletes locally. Cuckoo filters are used to approximate files that are associated with a given keyword within a node.

3.1 Multi-keyword Searching Options

There are two different searching options for our file system that utilizes Cuckoo filters to save network bandwidth, explicit and throughput. The default search option is the throughput multi-keyword search because it puts the least

amount of traffic on the network. This ensures that a very large keyword search does not occupy most of the network traffic within the system which allows for more searches to be run simultaneously.

3.1.1 Explicit Multi-keyword Search

The explicit multi-keyword search returns all files associated with the set of keywords. This search guarantees that all files returned are associated with all keywords. The explicit search for n keywords requires $2n$ messages between n servers involved. Every server must review the search results a second round to remove to eliminate false positives introduced by each filter. At the end of the search, all files are sent to the client. The complete algorithm is displayed in Figure 3.1 and explained below.

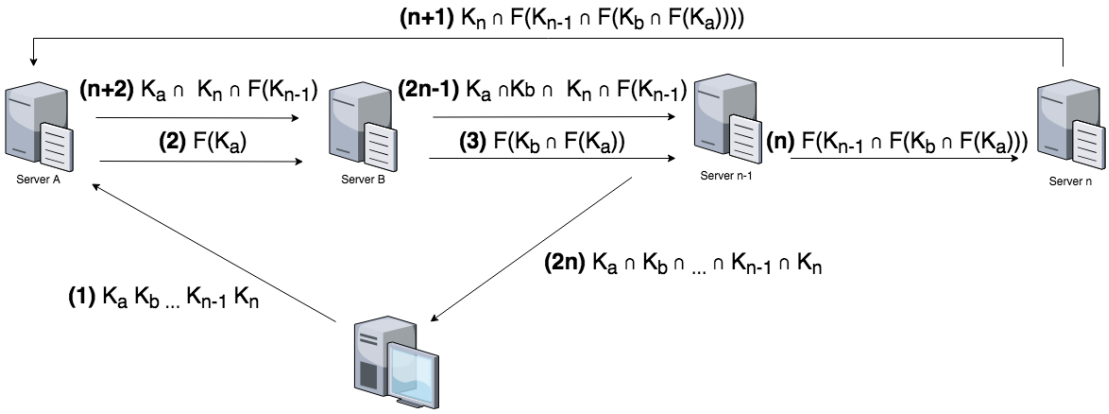

Figure 3.1: The Explicit Multi-keyword Search

The Explicit Multi-keyword Search returns **only** files associated with every

keyword within the search query. This correctness is at the cost of high latency due to the number of messages sent within the network per request. First the request is received by server S_a sends who send a Cuckoo filter $F(k_a)$ to approximate the files associated with k_a to server S_b. Server S_b sends the $F(k_b \cap F(k_a))$ to the next peer, this process repeats until S_n receives the Cuckoo filter. This can be referred of as phase one, where each server associated with a keyword receives a filter and relays the intersection of the received filter and its local filter for its keyword to the next until the every node for a given search receives the final Cuckoo filter. In phase two, the last server S_n sends a list of file IDs to the initial node that started the search algorithm. Phase two is where all false positives are eliminated by removing all files that do not intersect with the given list of file IDs. The search ends with server S_{n-1} replying to the host that initiated the search as seen in Figure 3.1.

3.1.2 Throughput Multi-keyword Search

The throughput multi-keyword search attempts to return only files associated with a given set of keywords with the potential of returning additional unrelated files. This search uses the least amount of bandwidth possible in a search. Only Cuckoo filters are sent within the network between peers. The known weakness with this algorithm is the throughput search results man contain files that are not

associated with all keywords due to false positives from the Cuckoo filters. This drawback can be overlooked by BEKSFSS users will still receive **all** correct files from their search.

This is coined as the Throughput Search because offers higher availability, lower latency and higher throughput in comparison to its efficient counter-part. The server that receives the search request does not need to compute any intersections and does not need to process any future requests for that specific search query. Each node that receives a request and hands off the request to the next node until each keyword has been searched for. Each server until the final server only needs to calculate and forwards its intersection of its files and the filter. Only the final server must check its own files against the final Cuckoo filter to determine the search results.

An example of the Throughput search of BEKSFSS is show in Figure 3.2. First the request is received by server S_a sends who send a Cuckoo filter $F(k_a)$ to approximate the files associated with k_a to server S_b. Server S_b sends the $F(k_b \cap F(k_a))$ to the next peer, S_{n-1}. This can be referred of as phase one, where every server that stores the files associated with the keyword receives a filter and relays a more robust filter to the next until the every node for a given search receives the final Cuckoo filter. After phase one, the final server responsible for the last word in the keyword search S_n responds to the host with the intersection

of its local files and the Cuckoo filter received by S_{n-1}.

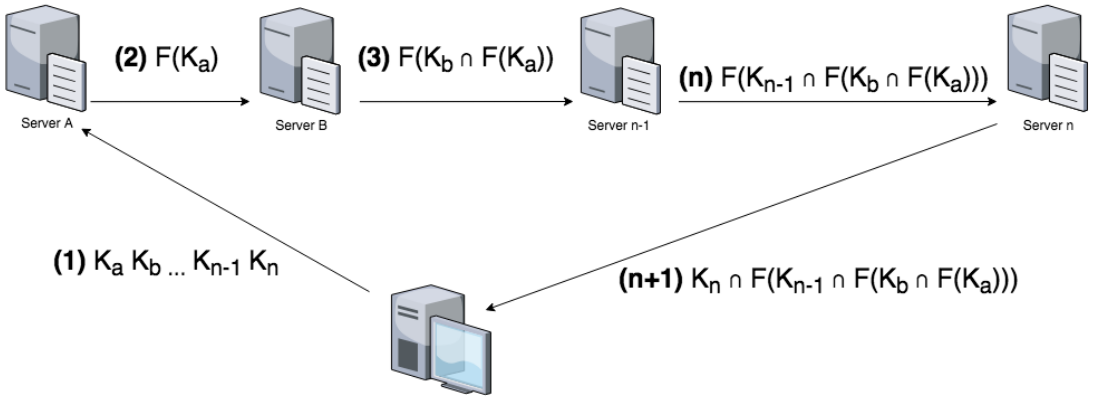

Figure 3.2: The Throughput Multi-keyword Search Algorithm

3.1.3 Implementation

Our simulation consisted of Docker containers running a simple flask application. The Docker containers run separately and were placed on the network by hashing thier unique network IP. We proposed to build an open system what allows a client to store, delete, and search for files based on keywords [4]. Each network node can receive client requests to add, delete, and search for files via HTTP requests. Each node holds Cuckoo filters for its own keywords for its files. Our implementation was written in Python 3.3 and had options to run any number of network nodes and hold any number of files with any number of keywords.

Our simulation used file names and keywords were generated from the top 10,000 most commonly used words in the English language. File data is not

necessary for this experiment because this study only investigates the inter-node traffic which does not include any file data. The file names are used as unique file identifiers to identify the accuracy of the search results.

Our goal in our file storage implementation is to show bandwidth saved, versus speed versus result accuracy. We tested the effects of the number of hosts within our network, cuckoo-filter bandwidth costs, and the accuracy of search results when using the high-throughput technique.

3.1.4 Results

We implemented BEKSFSS, a decentralized, highly distributed, peer-to-peer file storage system with a bandwidth efficient search algorithm. A client has the ability to interact with the file system without any complications any differently than a typical centralized data storage solution. Our system employs a chordal network model to partition all files across its nodes on its linked keywords.

We determined that an improvement in two or more word multi-keyword searches can significantly increase overall search latency and cost on network bandwidth. The necessity for multi-keyword searches was observed by trace searches from the IRC proxy cache system [13], a ten cache system across the United States. Across a ten-day period in January of 2002, their data revealed that 67.1% of all searches searched between the lengths of 2-5 keywords while 28.5% of the searches

were for single keywords [10].

3.1.4.1 Simulation Results

We expect the identification of correct index servers in $O(logn)$ messages based on Chord's lookup service. A client n-keyword search query takes a minimum of n inter-node messages contain only cuckoo filters. No file data is transferred between nodes. The bandwidth required for an search is constant and scales with number of keywords within the search and is completely separate from the size of the network.

As seen in Figure 3.3, the number of bytes sent between all network nodes is dependent on number of keywords within a search and not the number of nodes within the network. The prior work [10] used an Explicit Search with Bloom Filters which sends a list of file names to $k-1$ nodes for a search with k keywords. The Throughput search only sends Cuckoo filters between each host.

In Table 3.2, the theoretical values are based on the assumption that keywords have an average of 6.4 characters and the average filter size is 100 bytes.

As seen in Figure 3.4, throughput searches are significantly faster by at least 4 times faster than explicit searches made for the same exact files, queries, nodes, and network setup. The latency is significantly higher because an explicit search for n keywords requires $2n$ hops in comparison to a throughput search that only requires n hops. An explicit search also sends a list of file names between every

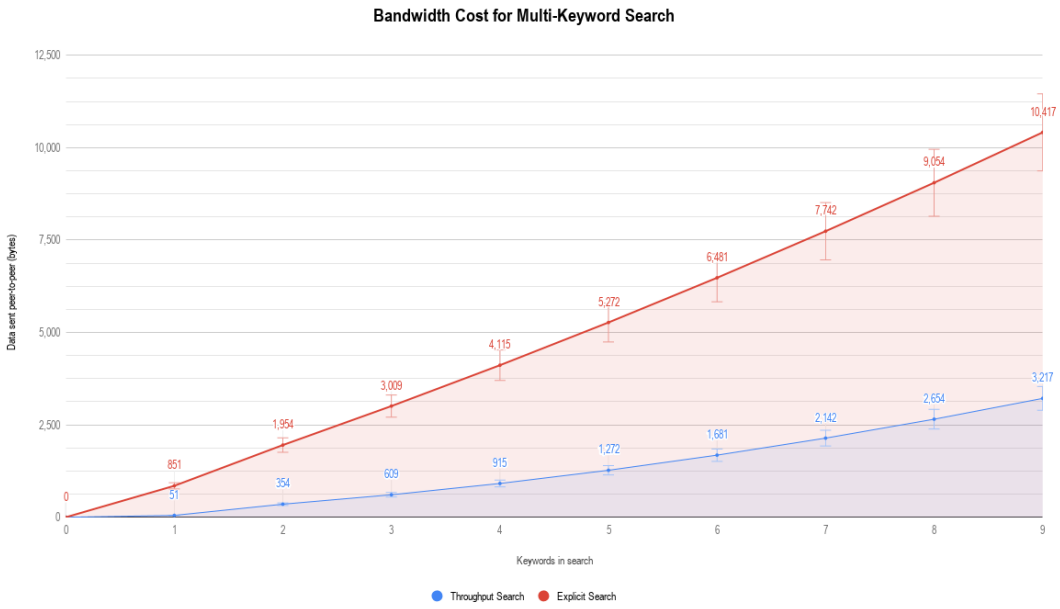

Figure 3.3: Network Bandwidth Cost for Multi-keyword Search

Keywords	**Bytes Exchanged**
1	51.49
2	354.49
3	608.98
4	914.97
5	1,272.45

Table 3.2: Network Traffic

node during the second phase of the search algorithm to eliminate false positives from the Cuckoo filters.

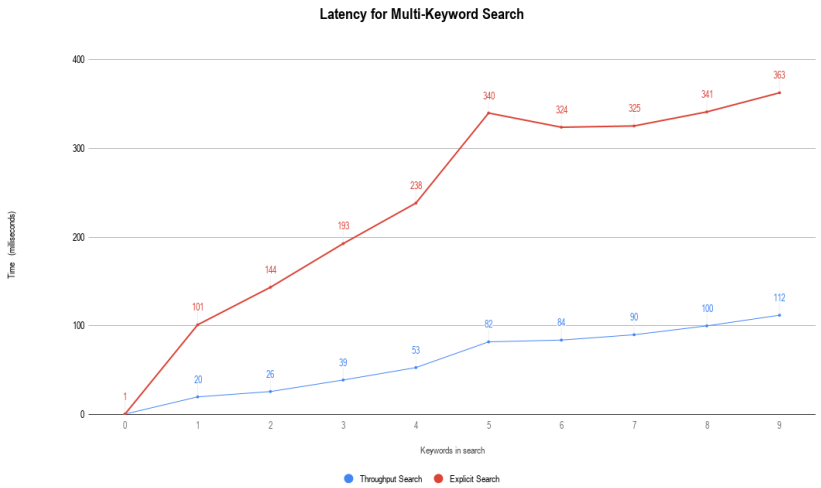

Figure 3.4: Latency for Multi-keyword Search

3.2 System Drawbacks

Our system only supports keyword searches and returns only the files that are related to the given all keywords within the search. The search results are narrowed down when more keywords are included.

Chapter 4

Conclusion

This paper presents a novel application of cuckoo filters in peer-to-peer systems to minimize total bandwidth within the network. Peer-to-Peer Bandwidth Efficient Keyword Search File Storage System proposed maintains P2P scalability and high availability while supporting file insertions, deletions, and searches. This simple P2P file storage solution has many applications. In summary, we were able to develop a purely decentralized file storage solution with a multi-keyword search algorithm with minimal network traffic.

Bibliography

[1] Gnutella. http://gnutella.wego.com/.

[2] Napster. https://www.napster.com/. Accessed: 2018-06-08.

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

[4] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30(1-7):107–117, 1998.

[5] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 75–88. ACM, 2014.

[6] George D. Greenwade. The Comprehensive Tex Archive Network (CTAN). *TUGBoat*, 14(3):342–351, 1993.

[7] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663. ACM, 1997.

[8] Michael Mitzenmacher, Salvatore Pontarelli, and Pedro Reviriego. Adaptive cuckoo filters. In *2018 Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 36–47. SIAM, 2018.

[9] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[10] Patrick Reynolds and Amin Vahdat. Efficient peer-to-peer keyword searching. In *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, pages 21–40. Springer-Verlag New York, Inc., 2003.

[11] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 329–350. Springer, 2001.

[12] Ion Stoica, Robert Morris, David Liben-Nowell, David R Karger, M Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-

peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking (TON)*, 11(1):17–32, 2003.

[13] D. Wessels and K. Claffy. The ircache project. https://www.ircache.net/. Accessed: 2018-06-08.

[14] Ben Y Zhao, Ling Huang, Jeremy Stribling, Sean C Rhea, Anthony D Joseph, and John D Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on selected areas in communications*, 22(1):41–53, 2004.