Evaluation and Application of Bloom Filters in Computer Network Security

By

Joseph Dzidefo Kwadwo Mensah Asempapa Agbeko

Submitted in Partial Fulfillment of the Requirements

For the Degree of

Master of Science

in

Mathematics

Youngstown State University

August 2009

Evaluation and Application of Bloom Filters in Computer Network Security

Joseph Dzidefo Kwadwo Mensah Asempapa Agbeko

I hereby release this thesis to the public. I understand that this thesis will be made available from the Ohio LINK ETD Center and the Maag Library Circulation Desk for public access. I also authorize the University or other individuals to make copies of this thesis as needed for scholarly research.

Signature:

Joseph Dzidefo Kwadwo Mensah Asempapa Agbeko, Student Date

Approvals

Dr. Graciela Perera, Thesis Advisor Date

Dr. Jamal Tartir, Committee Member Date

Dr. John Sullins, Committee Member Date

Peter J. Kasvinsky, Date
Dean of School of Graduate Studies & Research

ABSTRACT

Unstructured Peer-to-Peer (P2P) networks for content distribution are decentralized and robust. Searching for content in the network is based on the Gnutella Protocol. Broadcast Updates Look–up Search Protocol (BULLS) reverses Gnutella and enables a local look-up search at the cost of storing all the files shared in the network. In this thesis we introduce the use of bloom filters in the design and evaluation of a data structure that reduces space and search time in P2P networks based on BULLS. We also discuss the main ideas of a new Space Efficient Local Look–up Search (SELLS) protocol that is based on BULLS and uses this new data structure.

The new data structured is called the Inverse Bloom Filter (IBF) and uses bloom filters. A bloom filter is a space efficient probabilistic structure for membership queries. That is, they can be used to efficiently determine if a file is stored at a host. The cost is a small probability of error called false positive.

The challenge is to evaluate the search efficiency (i.e., remember searches that have not been successful) of bloom filters as the primary data structure of SELLS. The empirical evaluation can be achieved using real file names from a P2P network and determining the false positive rate of the bloom filter.

Novel applications of SELLS could possibly include secure key distribution; building block towards securing P2P networks

ACKNOWLEDGMENT

First and foremost, my heartfelt gratitude goes to God Almighty for the opportunity, protection, wisdom, guidance and mercies for having seen me through four years of my undergraduate studies.

My sincere gratitude and thanks go to my supervisor Prof (Dr.) Graciela Perera of the Department of Computer Science and Information System, Youngstown State University, Ohio, for her immeasurable suggestions, corrections, patience, enormous encouragement and excellent supervisory role during the entire process of this work. Again, my endless thanks go to my mother, Miss Dinah Enyo Klu,who did not see or used distance as an excuse but rather showed her love and support through diverse ways especially, prayers. To my siblings and in-laws I would like to say God bless you all for helping in one way or the other with my masters' education.

I cannot conclude by not expressing my profound gratitude to the following persons, Courtney Akosua Dunlap, Grace Marfo and Bismark Oduro for their encouragement and support.

Finally, my appreciation goes to my committee members Dr. John Sullins and Dr. Jamal Tartir for their support and encouragement.

TABLE OF CONTENTS

Abstract..iii

List of Tables... vi

List of Figures vii

Chapter 1: Introduction 1

Chapter 2: Background... 3

Chapter 3: A Space and Search Efficient Data Structure..................... 17

Chapter 4: Empirical Evaluation 27

Chapter 5: Conclusions 33
5.1 Future Work 34

References 35

LIST OF TABLES

Table 1: 12

Table 2: 13

Table 3, 4: 14

Table 5, 6: 20

Table 7: 28

Table 8: 29

Table 9: 30

Table 10: 31

Table 11: 32

LIST OF FIGURES

Figure 1: 4

Figure 2: 8

Figure 3: 17

Figure 4: 23

Figure 5, 6, 7: 24

Figure 8, 9: 25

Figure 10, 11: 26

Figure 12: 28

Figure 13: 29

Figure 14: 30

Figure 15: 31

CHAPTER 1

Introduction

Gnutella, a popular unstructured Peer–to-Peer (P2P) network distributes content (files) in a decentralized way which is self–organized, and robust [1]. Now due to the inefficient nature of Gnutella which generates much traffic overhead from query messages, it is of great importance to design a space efficient protocol with reduced traffic overhead so that P2P can be adopted and used on any device. In this paper the design and evaluation of a Space Efficient Local Look-up Search protocol, is studied by introducing Bloom Filters (BF).

A Bloom Filter is a simple, space-efficient, randomized data structure for representing a static set, in order to support approximate membership queries [3]. A Global Bloom Filter is one way in which a Bloom filter can be incorporated in a P2P network depending on whether the filter operations are performed before the tuples are transmitted to their destination node. One advantage with Global Bloom Filter (GBF) is that, it does not transmit unwanted tuples to remote nodes by performing the filter operation first, [2]. Therefore blooms filters can be used to space efficiently determine whether a file is stored in a network. The probability error of looking up for a file is called false positive. The question then is; How the empirical evaluation of the Space Efficient Local Look-up Search can be achieved? One way is by using real filenames used in a P2P network and determining the number of times successful searches occur.

In the light of this, the contributions made by this thesis are:

1) Explain the main ideas of a new Space Efficient Local Look-up Search (SEELS) protocol that includes the use of BF

1) Design and development a model for GBF.

2) Prototype implementation and empirically evaluation based on the rate of successful searches of GBF. Real file names from a P2P network are used for the evaluation.

3) The analytical probability of error for logical operations (AND and OR) were studied to deduce the analytical probability of error for the logical operation XOR. Logical operations (AND, OR, XOR) allow to compare the shared files between nodes in a P2P network.

The next section gives the background the P2P protocols used by this thesis for file searching and the theory of bloom filters.

CHAPTER 2

Background

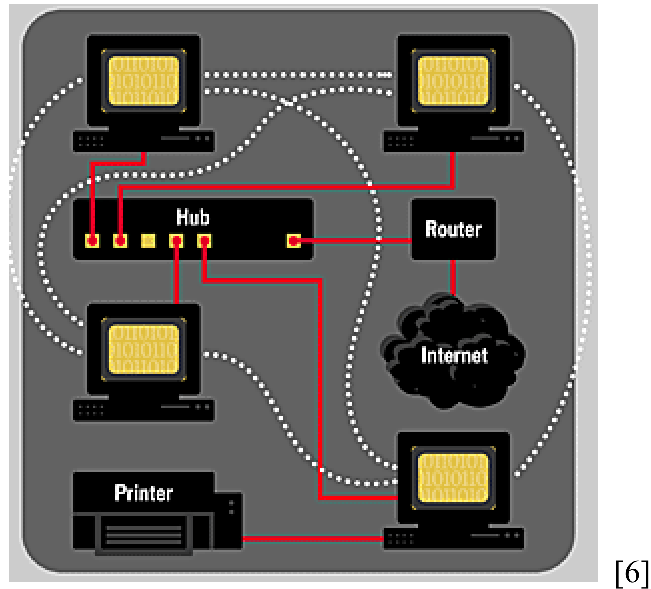
This chapter describes the three main ideas in which our work is based upon. The first is the summary of the Gnutella P2P network and how it generates query overhead traffic. The second idea is an improvement of Gnutella called BULLS that generates less traffic than Gnutella but may not be space efficient with respect to storage cost. The last idea is a probabilistic data structure called Bloom Filter that can reduce the storage cost of the BULLS protocol.

2.1 Gnutella a P2P Network

In its simplest form, a peer-to-peer (P2P) network is created when two or more PCs are connected and share resources without going through a separate server computer. A P2P network can be an adhoc connection—a couple of computers connected via a Universal Serial Bus to transfer files. Also a P2P network can be a network in which special protocols and applications set up direct relationships among users over the particular network. The most distinct characteristic of P2P computing is that there is symmetric communication between the peers; each peer has both a client and a server role. The advantages of the P2P systems are multi-dimensional; they improve scalability by enabling direct and real-time sharing of services and information; enable knowledge sharing by aggregating information and resources from nodes that are located on geographically distributed and potentially heterogeneous platforms; and, provide high availability by eliminating the need for a single centralized component. The basic

characteristic of the P2P network is that there is a group of nodes with the same type of interests connected over the same communication system.

The P2P network is self-organized and self-administrative as the nodes autonomously discover their peers, and self-healing as the nodes automatically try to find new peers if their current peers are (temporarily or permanently) disconnected from the network.



[6]

Figure 1. P2P Network

Figure 1 shows how a P2P network operates. The solid lines indicate physical, hard-wired network cables. The dotted lines indicate that each PC can communicate and share files with every other PC on such a network. A printer attached to one PC can be used by other PCs on the network that is if that printer's PC allows such use.

Current peer-to-peer applications are used for sharing resources, such as music files or video clips, and consecutively support only rudimentary search mechanisms. Typically the user specifies the name of the file he/she is looking for and searches for the file using a mechanism, which we will talk about in the next sections to come. Clearly, to support

higher level resources (such as, databases) for more sophisticated applications more efficient query mechanisms are required.

Users in a Gnutella file sharing P2P network search for files by flooding the network with queries. A file search requires the user to know the entire name of the file searched for or a sub string contained in the file name. Queries in Gnutella are thus sub-string searches over file-names. However a user at a node cannot determine what file is been shared in the network. That is, it is not possible to have knowledge of the entire set of files shared in the network. Also searching for multiple targeted files with no common sub-strings in their file-name, a query for each file must be made.

Hence searches are not efficient because you need to broadcast queries all the time. Additionally, queries that are unsuccessful are not remembered causing the network overhead traffic with respect to queries.

The next section talks about BULLS which is an improvement of Gnutella which generates less traffic than Gnutella at the expense of the storage space required to store all the shared files in the network.

2.2 Description of BULLS

BULLS is a P2P Gnutella based protocol that reverses the broadcast search paradigm and explores broadcasting file updates instead of queries. Figure 2 is the BULLS Finite State Machine (FSM) based on Gnutella protocol version 0.6 (Gnutella version 0.6). The FSM

shown in Figure 2 only describes the behavior of an ultra peer host. Ultra peer hosts, and not leaf hosts, exchange overhead messages (i.e., query and query hit messages). The behavior of a leaf host in BULLS is the same as in Gnutella; that is, only generating query message overhead traffic. The query hit message response from an ultra peer to a query message from one of its leaves is omitted from the FSM because it does not impact the overall overhead query hit traffic. In addition, the data structure used by BULLS is only stored by ultra peer hosts. Each ultra peer host stores in the data structure its own share file listing (set of shared files) and the shared file listing of the leaf hosts connected to it. The four states for the improved version of BULLS with ultra peers are defined as in [1, 7]. They are **INITIALIZE**, **IDLE**, **SEARCH**, and **SELECT**. The detailed description of the states and transitions is given below:

- **INITIALIZE**: An ultra peer host entering the network can be in this state by requesting to receive neighbor addresses of ultra peers (neighbors) or leaves and downloading the data structure from a specialized bootstrapping host. On the reception of a response with the requested neighbor addresses, the ultra peer host connects to its neighbors, forwards its own shared file list (one update message per file shared) and the share file listing of the leaves (one update message per file shared) to ultra peer neighbors (neighbor host that are ultra peers) only and transitions to **IDLE,** [1, 7].

- **IDLE**: In this state an ultra peer host can 1) make a file search by a local look-up in the data structure and transition to **SEARCH**, 2) detect a change in the data structure, repeat to ultra peer neighbors via an update message the changes in data

structure (one update per change) and remain in **IDLE**, 3) receive an update message, modify the data structure with the update received, store it in the cache, repeat it (send update message to all ultra peers neighbors except the one from which the message was received), and remain in **IDLE**, 4) receive a query message from a leaf host, repeat the query to all of its ultra peer neighbors, and remain in **IDLE**, 5) receive a depart message, update data structure by modifying departing host's row entry and repeat depart message to ultra peer neighbors, or 6) disconnect from the network by sending a depart message,[1,7].

- **SEARCH**: In this state the ultra peer host waits for results from a local look-up and it can 1) transition to **SELECT** if local look-up is successful or 2) transition to **IDLE** if local look-up does not return results, [1, 7].

- **SELECT**: In this state a host is selected from which to download a file (the host can be an ultra peer or a leaf). The set of possible hosts to select from is returned by the successful local look-up executed in the **SEARCH** state. The ultra peer host downloads the file, updates its shared files, updates its data structure, and transitions to **IDLE,** [1,7].

The transitions that impact the amount of overhead traffic generated are the same five transitions that impact the overhead traffic in [1]. The transitions that impact the amount of overhead traffic cause the broadcast of the shared file list and the broadcast of updates when the shared file list is modified, that is when a file is added, deleted or downloaded.

Also, the data structure used by the improved BULLS remains the same as described in [1].

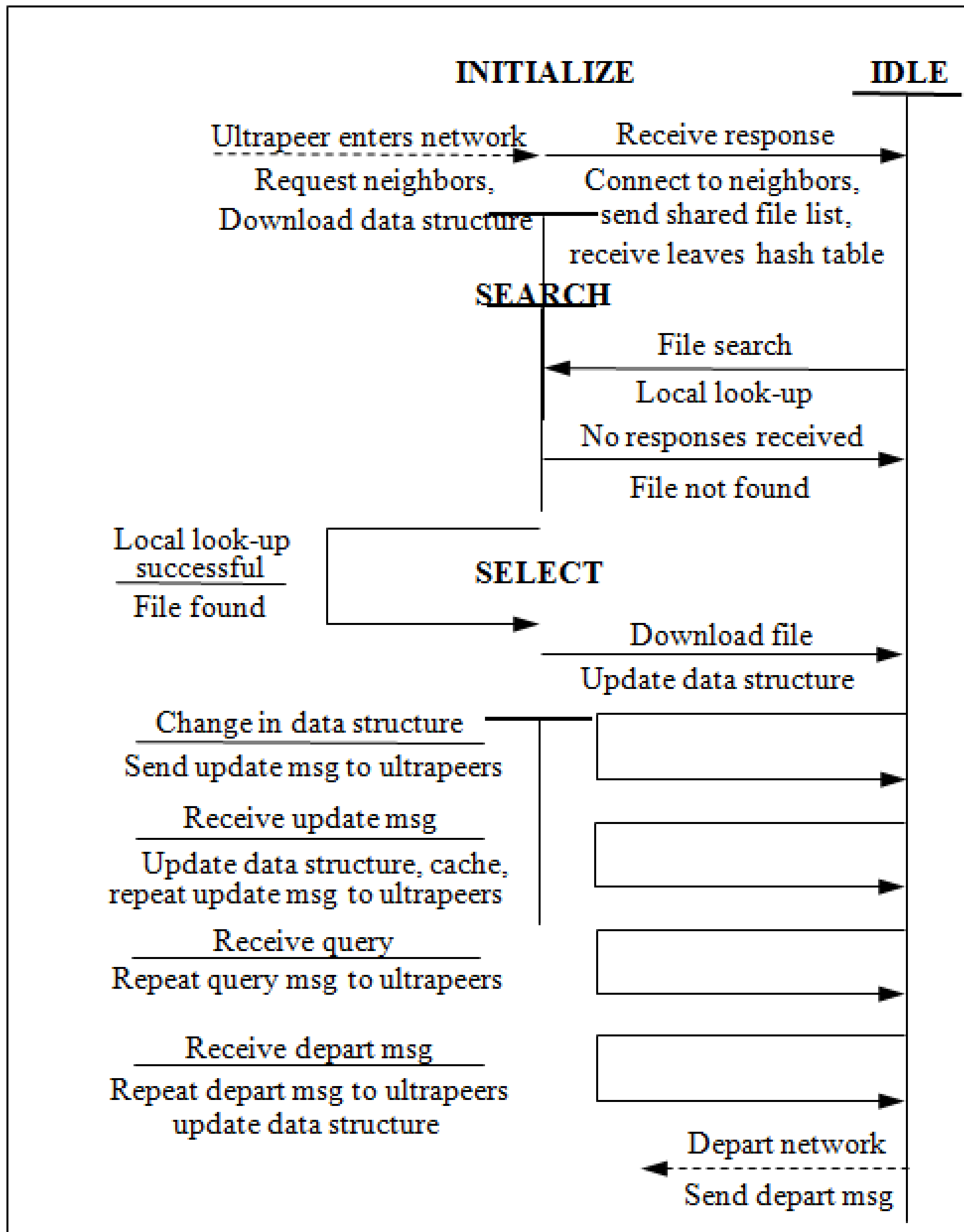


Figure 2.BULLS FSM based on Gnutella version0.6

The next section talks about the last idea which is a probabilistic data structure called Bloom Filter that can be used to reduce the storage cost of the BULLS protocol.

2.3 Bloom Filter

Bloom filters allow the data structure used by SELLS to be more space efficient than BULLS. A bloom filter is therefore a simple, space-efficient, randomized data structure for representing a static set, in order to support an approximate membership queries. It was formulated by Burton H. Bloom in 1970, [3].

A bloom filter for representing a set $S = \{S_1, S_2, \ldots S_n\}$ of n elements is described by an array of m bits, with all bits initially set to 0. A bloom filter uses k independent hash functions $h_1, h_2, \ldots, h_k$ ranging over {1,..., m}. The hash functions map each item in the universe to a random number uniform over the range {1... m}. For each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$. To check whether an element y belongs to S, one just needs to check whether all the $h_i(y)$ bits are set to 1. If so, then y is a member of S, although this could be wrong with some probability. Otherwise, we assume that x is not a member of S, [4].

This situation is called a false positive and the probability that this occurs is called the false-positive rate. Hence, a bloom filter may yield a false positive, for which it suggests that an element x is in S even though it is not. The probability of a false positive for an element not in the set can be calculated in a straightforward fashion, given our assumption that hash functions are perfectly random, [4].

Assume that a hash function selects each array position with equal probability. The probability that a certain bit is set to 0 by a certain hash function during the insertion of an element is then;

$$1-\frac{1}{m} \quad \text{.. (2.3.1)}$$

The probability that it is not set by any of the hash functions is;

$$\left(1-\frac{1}{m}\right)^{k} \quad \text{.. (2.3.2)}$$

The probability that a certain bit is still 0 is;

$$\left(1-\frac{1}{m}\right)^{kn} \quad \text{.. (2.3.3)}$$

The probability that it is 1 is therefore;

$$1-\left(1-\frac{1}{m}\right)^{kn} \quad \text{..................................... (2.3.4)}$$

Each of the *k* array positions computed by the hash functions is 1 with a probability as above. The probability of all of them being 1, which would cause the algorithm to erroneously claim that the element is in the set, is then;

$$\left(1-\left(1-\frac{1}{m}\right)^{kn}\right)^{k} \approx \left(1-e^{-kn/m}\right)^{k} \quad \text{........ (2.3.5)}$$

The probability of false positives decreases as k (the number of hash functions) increases, and increases as n (the number of inserted elements) increases. For a given *m* and *n*, the value of *k* (the number of hash functions) that minimizes the probability is

$$\frac{m}{n}\ln 2 \approx \frac{9m}{13n} \approx 0.7\frac{m}{n} \quad \text{................................. (2.3.6)}$$

which gives the false positive probability of

$$\left(\frac{1}{2}\right)^k \approx 0.6185^{m/n}$$ …………………………………… (2.3.7)[4]

This proof which has appeared in many papers throughout the years has also been cited by other papers as incorrect. Such papers went ahead by explaining that the error occurs in deriving (2.3.5), where there is an implicit assumption that the event "$h_1(y_1) = 1$" and the event "$h_2(y_2) = h_3(y_3) = \cdot\cdot\cdot = h_{i-1}(y_i) = 1$" are independent[4]. At first glance, this seems to be true, since $y_1,\ldots,y_i$ are independent. Thus, the paper suggests that the analysis of the false-positive rate is not nearly as straightforward as one would hope and further went ahead by giving an exact formula for the false-positive rate of Bloom filters as a theorem;

Theorem *Let $p_{k,n,m}$ be the false-positive rate for a Bloom filter that stores n elements in a bit-vector of size m using k hash functions. Then,*

$$P_{k,n,m} = \frac{1}{m^{k(n+1)}} \sum_{i=1}^{m} i^k i! \binom{m}{i} \left\{ {kn \atop i} \right\}$$ ………………….. (2.3.8)[4]

Unfortunately, this formula is not anywhere near closed form, but could be useful for small values of k, n and m.

The next section describes a closed form for the false positive rate of applying the basic logical operations between Bloom Filters. These operations are useful to summarize two bloom filters ("OR"); which includes all elements between the two bloom filters and the ("AND"); which obtains the common elements between the two bloom filters.

2.3.1 False Positive Rates for two logical operators

Because bloom filters are compact probabilistic representations of sets it is interesting to study the two the logical operations "AND" and "OR" that correspond respectively to the basic set operations of union and intersection. Many applications using bloom filters require these operations as they can be used to easily obtained and compare all elements of two bloom filters at the expense of a false positive rate.

For "AND" (&), let p and q be two bloom filters, containing files to be searched for each of size n bits. Now the table returns Truth (T) when the particular file been searched for is in both bloom filters p and q. The table returns False (F) when it is otherwise, that is when a particular file is in one of the bloom filters but not the other or if the file is not found in both bloom filters. So the truth table can be written as;

Table 1

p	q	p&q
T	T	T
T	F	F
F	T	F
F	F	F

[2]

Now it can be seen that $p\&q$ is only T if both p and q are T. Assuming that elements in a bloom filter represent files names of a P2P network. Thus, each P2P node keeps in a bloom filter the set of files it shares. The only way for which $p\&q$ is T is when two given P2P nodes are sharing the same set of files. For the purpose of this research, we denote T=1 and F=0. This implies Table 1 becomes the table below;

Table 2

p	q	p&q
1	1	1
1	0	0
0	1	0
0	0	0

[2]

The above logical table is used for the proofs that determine the false positive rate of the OR and the XOR latter on. Now let S1 and S2 be sets respectively. Assume that, the ith bit will be set in two filters p and q, by some element S1 in and S2 that is (S1 $\cap$ S2) or element in S1 but not in S1 and S2i.e{ S1 -(S1 $\cap$ S2)} and another element in S_2 but not in S_1 and S_2 that is{ S_2 -($S_1 \cap S_2$)}. Then the probability that the ith bit is set to both filters is therefore;

Prob. (set to 1 in both sets) +Prob. (set to zero) * Prob (set to 1 in set1 and set 2).This implies that we have;

$$\left(1-\left(1-\frac{1}{m}\right)^{kn}\right)+\left(1-\frac{1}{m}\right)^{kn} * \left(1-\left(1-\frac{1}{m}\right)^{kn}\right)\left(1-\left(1-\frac{1}{m}\right)^{kn}\right).$$

$$=\left(1-\left(1-\frac{1}{m}\right)^{k|s_1\cap s_2|}\right)+\left(1-\frac{1}{m}\right)^{k|S_1\cap S_2|}\left(1-\left(1-\frac{1}{m}\right)^{k|S_1-(S_1\cap S_2)|}\right)\left(1-\left(1-\frac{1}{m}\right)^{k|S_2-(S_1\cap S_2)|}\right)$$

$$m\left(1-\left(1-\frac{1}{m}\right)^{k|S_1|}-\left(1-\frac{1}{m}\right)^{k|S_2|}+\left(1-\frac{1}{m}\right)^{k(|S_1|+|S_2|-|S_1\cap S_2|)}\right)^{k} \quad ......(2.3.9)[3]$$

Hence we can find the estimate of the false positive rate of the "AND" given, $|S_1|,|S_2|$, k, m and the magnitude of the inner product. Also we can estimate $|S_1|, and |S_2|$ by counting number of 0 bits in the Bloom filters for S_1 and S_2, if $|S_1|, and |S_2|$ are not given.

For "OR", let *p* and *q* be two filters as described earlier. Then for the truth table for "OR" we have

Table 3

p	q	p|q
T	T	T
T	F	T
F	T	T
F	F	F

[2]

Now, it can be seen that *p*|*q* is T whenever either *p* is T, *q* is T or both *p* and *q* are T. For the purpose of this research, T=1, and F=0. Thus table 3 becomes table 4 below;

Table 4

p	q	p|q
1	1	1
1	0	1
0	1	1
0	0	0

[2]

Suppose there are two Bloom filters representing sets S_1 and S_2 with the same number of bits. Now using the same hash functions and taking the "OR" of the two bit vectors of the original Bloom filters will be the union of the two sets. Assuming that the jth bit will be the set in two filters *p* and *q* by some element in S_1 or S_2 i.e. $(S_1 \cup S_2)$. Then the probability that the jth bit is set to either filters will be;

Prob.(set to 1 in both sets) +Prob.(set to 1)**Prob*.(set to 0 in set1 or set 2)+*Prob*.(set to 0 in both sets).

$$\Rightarrow \left(1-\left(1-\frac{1}{m}\right)^{kn}\right)+\left(1-\left(1-\frac{1}{m}\right)^{kn}\right)\left(\left(1-\frac{1}{m}\right)^{kn}\left(1-\frac{1}{m}\right)^{kn}\right)+\left(1-\frac{1}{m}\right)^{kn}$$

$$=\left(1-\left(1-\frac{1}{m}\right)^{k|S_1\cup S_2|}\right)+\left(1-\left(1-\frac{1}{m}\right)^{k|S_1\cup S_2|}\right)\left(\left(1-\frac{1}{m}\right)^{k|S_1\cup S_2|}\right)+\left(\left(1-\frac{1}{m}\right)^{k|S_1\cup S_2|}\right)$$

$$=1-\left(1-\frac{1}{m}\right)^{k|S_1\cup S_2|}+\left(1-\frac{1}{m}\right)^{k|S_1\cup S_2|}-\left(1-\frac{1}{m}\right)^{k|S_1\cup S_2|+k|S_1\cup S_2|}+\left(1-\frac{1}{m}\right)^{k|S_1\cup S_2|}$$

$$=1-\left(1-\frac{1}{m}\right)^{k|S_1\cup S_2|}-\left(1-\frac{1}{m}\right)^{k|S_1\cup S_2|}+\left(1-\frac{1}{m}\right)^{k|S_1\cup S_2|}$$

$$\left(1-\left(1-\frac{1}{m}\right)^{k|S_1\cup S_2|}\right)^k \quad \text{..... (2.3.10) [3]}$$

In the section above we have demonstrated the impact that the hash functions have in the false positive rate of bloom filters. The next section briefly discusses what a hash is.

2.3.2 Importance of Hash Functions

A hash function is a partition function that maps a large domain of key values into a much smaller domain of hash values. In other words, let U be the universe of possible keys for a set of elements. It is generally assumed that all elements have unique integer

keys. Let *m* be the size of our array that will hold the elements. A hash function h(key) is a function that maps U to Z_m, [5].

Typical hash functions perform computation on the internal binary representation of the search-key and are mostly used to speed up table look-up or data comparison tasks such as finding items in a database or similar records in a large file.

A hash function may usually map two or more keys to the same hash value. In a sense this results in the collision of files been mapped which means that the hash function must map the keys to the hash values as evenly as possible, [5]. Based on this principle, we can say that to be able to effectively and locally look up files in a global data structure in a network, we may need to employ the techniques of hashing using Global Bloom Filters (GBF) where one would want to find if there is a given file or similar file. If we can then define a one-to-one mapping from U to Z_m, then h(k) is called a perfect hashing function.

CHAPTER 3

Space and Search Efficient P2P network

This section describes the main ideas of a new protocol called SELLS that incorporates the use of Bloom Filters in file searches. We first describe the SELLS protocol for P2P networks and then the design and evaluation of how SELLS reduces search time.

In particular we present the analysis, design and performance evaluation of using bloom filters as an alternative data structure for P2P file sharing applications.

3.1 The SELLS Protocol

The behavior of the SELLS protocol is very similar to the BULLS protocol behavior described in Figure 2. SELLS differs from BULLS in the data structure used to store the share file listing (the files shared by each host). SELLS uses a bloom filter to represent the list of shared files by each ultra peer host and can be used to performed file look-ups in the bloom filter to obtain the list of IP address or hosts from which the desired file can be downloaded,[1,7].

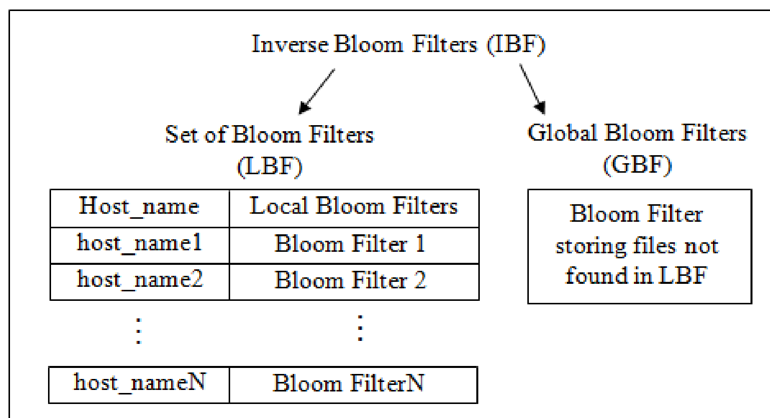


Figure 3.SELLS data structure [1,7]

Bloom filters allow the data structure used by SELLS to be more space efficient than BULLS. SELLS uses a compact probabilistic representation or bloom filter for all the files shared by each host instead of representing each file as a separate item. Also, a global bloom filter is used to filter searches with low probability of success. The SELLS's data structure is called an Inverse Bloom filters (IBF). The two components that make the IBF are a set of Local Bloom Filters (LBF) and a Global Bloom Filter (GBF). The IBF used by SELLS is described in Figure 3 and explained in detail below, [1,7]:

LBF: The first component shown in Figure 2 is a table with each row representing the data stored for a host in the network. The columns of the table in Figure 3 are the two basic types of data stored. The first column is the host-name and it is used to identify a host in the network (IP address or host identification number). The second column is the bloom filter representing the file share listing (set of file-names shared) of a host, [1,7].

GBF: The second component shown in Figure 3 is a bloom filter constructed by storing the file-names of unsuccessful searches. That is, after a file-name has been searched in LBF if the search is unsuccessful it will be added to GBF. This will create a bloom filter that will store the improbable files to be found in the network or the files that are not stored by any host in the network, [1,7].

When a SELLS host inserts a file in the file shared list, the corresponding bloom filter will be updated by setting the bits obtained from hashing the name of the file inserted. Contrary to a file being inserted in the file shared list, the deletion of a file causes the

bloom filter to be reconstructed. That is, each element in the file shared list is inserted in a new bloom filter that will substitute the first one [1, 7].

Additionally, because SELLS stores the shared file list as a single item instead of separate items, when an ultra per host enters the network it behaves different from BULLS.

That is, it forwards its own shared file list (one update message or one bloom filter per shared file list) and the share file listing of the leaves (one update message or one bloom filter shared file list) to ultra peer neighbors (neighbor host that are ultra peers) only and then transitions as BULLS does to the IDLE state.

To detect whether an update is a deletion or an insertion of a file, we can use the logical operator XOR and AND. That is, the AND already discussed in the previous section can be used to detect the inserted elements in a bloom filter. On the other the hand, the XOR logical operator can be used to determine the bits that have changed between two bloom filters. Given two bloom filters b1 and b2 assuming that b2 is an updated bloom filter of b1, we can determine if the bits that have change between the bloom filter by a combination of the XOR and AND.

Applying XOR over b1 and b2 yield the bits that have changed between the bloom filters. Let b3 be the bloom filter resulting from b1 XOR b2 then b3 AND b1 will yield the elements that are deleted from b1. In particular the false positive rate of the XOR for two bloom filters is shown below;

For the eXclusive “OR” (“X”), let *p* and *q* be two bloom filters, containing files to be searched for each of size *n* bits. Now the table returns Truth (T) when the particular file been searched for is not in that particular bloom filter and the table returns False (F) when it is otherwise.

So the truth table can be written as;

Table 5

p	*q*	pXq
T	T	F
T	F	T
F	T	T
F	F	F

[2]

Now,the XOR is T if either p is T or q is T, but not both, as we can see from the Table 5.

For our research denote T=1 and F=0. Hence Table 5 becomes table 6 below;

Table 6

p	*q*	pXq
1	1	0
1	0	1
0	1	1
0	0	0

[2]

Now, suppose that there are two Bloom filters representing sets S_1 and S_2 with the same number of bits and using the same hash functions. Then taking the OR" and the "AND of the two bit vectors of the original Bloom filters will be the union and intersection of the two sets. Hence combining (A) and (B), we have;

$$= m\left(1-\left(1-\frac{1}{m}\right)^{k|S_1|}-\left(1-\frac{1}{m}\right)^{k|S_2|}+\left(1-\frac{1}{m}\right)^{k(|S_1|+|S_2|-|S_1\cap S_2|)}\right)^k + \left(1-\left(1-\frac{1}{m}\right)^{k|S_1\cup S_2|}\right)^k$$

$$m\left(2-\left(1-\frac{1}{m}\right)^{k|S_1|}-\left(1-\frac{1}{m}\right)^{k|S_2|}+\left(1-\frac{1}{m}\right)^{k(|S_1|+|S_2|-|S_1\cap S_2|)}-\left(1-\frac{1}{m}\right)^{k|S_1\cup S_2|}\right)^k \quad \text{... (3.1.1)}$$

With the false positive rate of the XOR we can calculate the false positive rate of detecting if an update has an insertion or a deletion. In the next section a model of GBF is developed and analyzed.

3.2 Model of a Bloom Filter

The design of the BF was based on the number of elements in the Bloom Filter denoted by *n*, the bloom filter size *m* (in bytes) and the false positive rate *fpr* for the optimal *k* of the BF. For a P2P network using SELLS, each element represented in the BF corresponds to a file-name. Thus, the number of elements in BF corresponds to the number of files a host shares. We present below an analysis of the relationship between BF size and *fpr* positive rate.

We assumed:

- Each file-name in the BF had 50bytes of space.
- The maximum number of files shared by the host is 1000
- Finally that a "good" *fpr* will be less than 1%.

However, in case a file is not found in the BF, we can use Bloom Filter to represent the set of files shared in the network and search for that particular file. In BULLS the cost of space in bytes of storing the average set of files stored by a host is 50bytes (400 bits). Using Bloom Filters we can reduce this cost by a certain percentage.

Based upon our assumptions in previous section, the next section talks about our design and implementation of the BF model and its implementation.

3.2.1 Design and Implementation of a BF

Bloom filters allow you to perform membership tests in just a fraction of the memory needed to store a full list of files. However the savings in space comes at a price in a sense that you run an adjustable risk of false positives, and one cannot remove a file from a filter once it is added. But in the many cases where those constraints are acceptable, a Bloom filter can make a useful tool. With the two components of a Bloom filter already mentioned, (that is a set of k hash functions and a bit vector of a given length) we choose the length of the bit vector and the number of hash functions depending on how many files we want to add to the set and how high an error rate we are willing to put up with.

We further went ahead to configure all of the hash functions in a Bloom filter so that their range matches the length of the bit vector. For example, if a vector is 400 bits long, the hash functions return a value between 1 and 400. We therefore designed and implemented the BF prototype by using *Java* as the programming language so that it can

be used on all platforms. We called the library Simple Bloom Filter (SBF). The description of the class used for implementing the experiment is a follows;

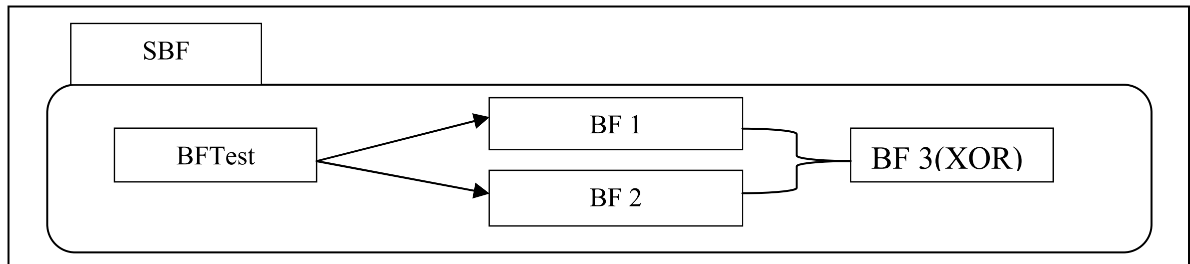


Figure 4.Description of GBF implementation

We implemented the prototype BF in one class and developed a test class to evaluate the BF prototype.

SimpleBloomFilter Class

The class constructs a SBF set. The class specifies the number of bits in the BF, and also specifies the number of items it expects to add. The class uses the latter to choose some optimal internal values to minimize the false-positive rate (which can be estimated with expected False Positive Rate ()).

The class uses the parameters;

- bitArraySize -The number of bits in the bit array (often called 'm' in the context of bloom filters)
- expectedElements- The typical number of items you expect to be added to the SimpleBloomFilter (often called 'n').

Some of the important class methods are given below:

- *fill* method fills the BF with *n* number of integers of size *m* bytes

```
SimpleBloomFilter fillbf(SimpleBloomFilter4<String> a, int start, int size) {
    for (int i = start; i < size; i++)
            {
            String val = String.valueOf(i);
     a.add(val);
            }
    return a;
  }
```

Figure 5. Filling Algorithm

- *membership* method adds members to the bloom filter with *n* number of integers of size *m* bytes

```
SimpleBloomFilter4 membership(SimpleBloomFilter<String> a, int start, int size)
{
    for (int i = start; i < size; i++)
            {
            String val = String.valueOf(i);
     if (a.contains(val)) System.out.println(val + " is member");
            a.add(val);
            }
    return a;
  }
```

Figure 6. Membership Algorithm

- *uniquemembership* method looks for unique members in the bloom filter that are integers

```
int uniquemembership(SimpleBloomFilter4<String> a, Set<String> b) {
    int b3unique = 0;
          Iterator iter = b.iterator();
    while (iter.hasNext()) {
      String elem = (String) iter.next();

            if (a.contains(elem))
                    {
                      System.out.println(elem + " is member");
                            b3unique = b3unique + 1;
        }
            }
    return b3unique;
  }
```

Figure 7. Unique Membership Algorithm

➢ *efp* method calculates the expected false positive (*efp*) rate for the BF.

```
int efp(SimpleBloomFilter4<String> a, Set<String> b) {
   int badele = 0;
         Iterator iter = b.iterator();
   while (iter.hasNext()) {
     String elem = (String) iter.next();

          if (a.contains(elem))
                {
                   System.out.println(elem + " is bad member");
                         badele = badele+ 1;
      }
         }
   return badele;
 }
```

Figure 8. Expected False Positive Rate Algorithm

Now with the class constructed, the next section describes the implementation of the BF class using the BF Test as given below:

1) Now we test to see if Bloom Filter 1 successfully remembers what was added

```
mammals.setNumBloomFilter(1);
 System.out.println("Bloom Filter 1 has: ");
 membership(mammals,1,10);
 System.out.println('\n');
 fill(bad,1,100);
System.out.println(" bad..." );
 System.out.println(efp(mammals,bad)- b1.size());
```

Figure 9.BF 1 Test

This empirically estimates false positive rate. It loops over a list of numbers. If *i* is a file name of bloom filter, add one to file of bad files, and then end the loop. The false positive rate is bad elements/ size of interval.

2) Now we test to see if Bloom Filter 2 successfully remembers what was added

```
mammals.setNumBloomFilter(2);
  System.out.println("Bloom Filter 2 has: ");
  membership(mammals,1,10);
  System.out.println('\n');
```

Figure 10.BF 2 Test

3) Now we test to see what Bloom Filter 3 has

```
mammals.setNumBloomFilter(3);
System.out.println("Bloom Filter 3 has: ");
  uniqueb3 = uniquemembership(mammals,b3);
  System.out.println('\n');
```

Figure 11.BF 3 Test

We based our developed prototype class from the one developed by Ian Clarks idea of a Bloom filters. For his implementation he used byte to represent a Boolean in a Boolean array. This means that for every actual bit stored, there are seven wasted bits – hence one might not be saving memory as you are supposed to.

In the next chapter we present the results gathered from the use of the implementation described before.

CHAPTER 4

Empirical Evaluation

This section describes the results obtain from the implementation of the Bloom Filter used by SELLS. The empirical results for the bloom filters are shown and compared to the analytical results obtained from the equations explained chapter 2.

4.1 Evaluation Metrics

The metrics used to evaluate the Bloom filter are;

(1) The size (in bytes) of files

(2) The number of file searches

(3) The false positive rate/ probability of error

These metrics aim to evaluate how robust and correct the Bloom filter implementation is. The probability of error can be understood as the ability to find a particular file in a network which is not supposed to be in that particular network for the number of files searched.

4.2 Empirical Results

The tables below show the results obtained from running our bloom filter implementation over an "average P2P node". We assume that the "average P2P node" shares at most 1000 files. Furthermore, to pick the size of the bloom filter we analyzed the size of each bloom filter for powers of two and powers of ten to study their differences.

Table 7 and Table 8 gives the empirical results for our prototype for bits size ranging for 32-16384bits and 400-4000bits with number of files been 1000 for each range of bits. These tables are also represented in graph form in figure 12 and figure 13

Table 7 M for 32-16384bits and N=1000

Bits Size	Number of files		Optimal K	fpr
M	**N**	**M/N**	**K=ln 2(M/N)**	**M**
32	1000	0.032	-2.7488	1.0000
64	1000	0.064	-2.0557	1.0000
128	1000	0.128	-1.3625	1.0000
256	1000	0.256	-0.6694	1.0000
512	1000	0.512	0.0237	1.0000
1024	1000	1.024	0.7168	0.6321
2048	1000	2.048	1.4100	0.3996
4096	1000	4.096	2.1031	0.2526
8192	1000	8.192	2.7963	0.1469
16384	1000	16.38	3.4892	0.0920

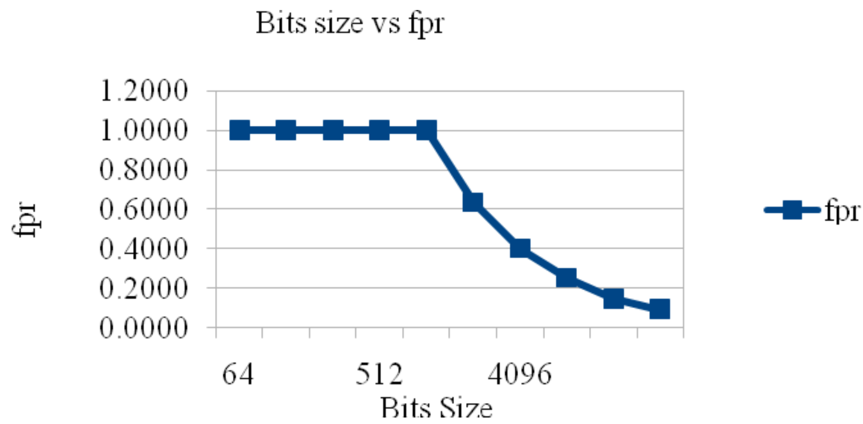


Figure 12. Bits Size vs. False Positive Rate

The results for the bloom filter size of 400-4000bits with N=1000 (shared files) is illustrated in table 8 and figure 13

Table 8

M for 400-4000bits and N=1000

Bits Size	Number of files		Optimal K	fpr
M	**N**	**M/N**	**K=ln 2(M/N)**	**M**
400	1000	0.4	-0.2231	1.0000
500	1000	0.5	0.0000	1.0000
600	1000	0.6	0.1823	1.0000
700	1000	0.7	0.3365	1.0000
800	1000	0.8	0.4700	1.0000
9000	1000	0.9	0.5877	1.0000
1000	1000	1	0.6931	0.6234
2000	1000	2	1.3862	0.3886
3000	1000	3	1.7917	0.1400
4000	1000	4	2.0794	0.0196

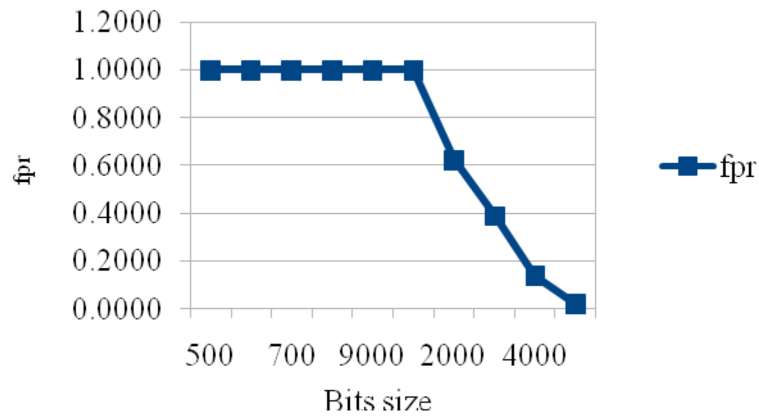


Figure 13. Bits Size vs. False Positive Rate

From the two tables and also from the graph, it could be seen that the fpr remained constant until the bits size was greater than or equal to the number of files been shared. With this observation, we went ahead with the analytical aspect of our results based on the formula stated in Chapter 2.

Table 9 and Table 10 gives the analytical results for our prototype for bits size ranging for 32-16384bits and 400-4000bits with number of files been 1000 for each range of bits. These tables are also represented in graph form in figure 14 and 15.

Table 9

M for 32-16384bits and N=1000

Bits Size	Number of files		Optimal K	fpr
M	**N**	**M/N**	**K=ln 2(M/N)**	**M**
32	1000	0.032	-2.7488	0.9847
64	1000	0.064	-2.0557	0.9697
128	1000	0.128	-1.3625	0.9403
256	1000	0.256	-0.6694	0.8842
512	1000	0.512	0.0237	0.7819
1024	1000	1.024	0.7168	0.6114
2048	1000	2.048	1.4100	0.3738
4096	1000	4.096	2.1031	0.1397
8192	1000	8.192	2.7963	0.0195
16384	1000	16.38	3.4892	0.0003

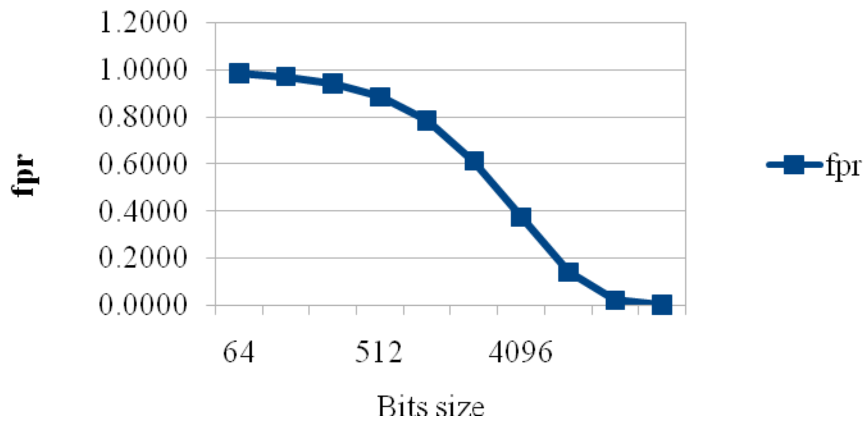


Figure 14. Bits Size vs. False Positive Rate

Table 10

Table 4.5 M for 400-4000bits and N=1000

Bits Size	Number of files		Optimal K	*fpr*
M	**N**	**M/N**	**K=ln 2(M/N)**	**M**
400	1000	0.4	-0.2231	0.8252
500	1000	0.5	0.0000	0.7866
600	1000	0.6	0.1823	0.7496
700	1000	0.7	0.3365	0.7144
800	1000	0.8	0.4700	0.6809
9000	1000	0.9	0.5877	0.6489
1000	1000	1	0.6931	0.6185
2000	1000	2	1.3862	0.3825
3000	1000	3	1.7917	0.2366
4000	1000	4	2.0794	0.1463

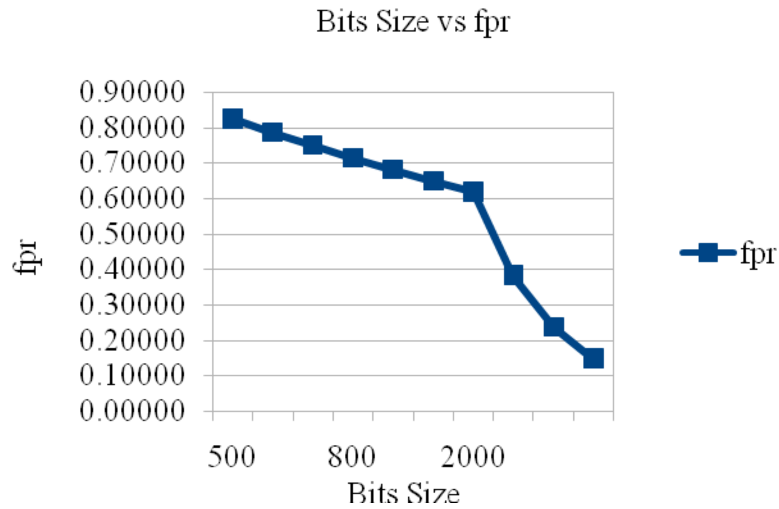


Figure 15. Bits Size vs. False Positive Rate

From the tables and graph above, we realized that the fpr started to decrease gradually until it got to an fpr of approximately 0.6 when it decreased suddenly in the 400-4000 Bits which was otherwise in the 32-16384 Bits. Overall, the analytical results more or

less matched the empirical results obtained from our implementation. Thus, the behavior of the BF model was validated.

Additionally, the comparison of the storage cost between the P2P protocol SELLS that uses GBF and the based protocol used by SELLS yields a 504% reduction. The comparison of the storage cost using the formulas in [1, 9] are shown in Table 11.

Table 11 Storage Space of BULLS vs GBF

Bits Size	Number of files	BULLS	GBF	% of Reduction
50	100	1010000	201000	502.5%
50	200	404000	80200	503.7%
50	300	909000	180300	504.2%
50	400	1616000	320400	504.4%
50	500	2525000	500500	504.5%
50	600	3636000	720600	504.6%
50	700	4949000	980700	504.6%
50	800	6464000	1280800	504.7%
50	900	8181000	1620900	504.7%
50	1000	101000000	20010000	504.7%

The next chapter present the conclusions and future work of the investigations described in this thesis.

CHAPTER 5

5. Conclusions

The investigation done in this thesis covers the analysis, design, implementation and evaluation of a new space and search efficient data structure for P2P networks.

Overall, from the results obtained from both the empirical and the analytical, we observed that there was negligible difference between the two results. That is, the maximum difference between the false positive rates (i.e. probability of error) was negligible, which in our opinion can be improved at the expense of a larger storage cost.

In particular the main results and conclusion from our work are the following:

1. Designed a model of space efficient data structure called using bloom filters.
2. Explained the use of BF in a P2P protocol called SELLS. BF can reduce the storage cost by 500% when compared to the based protocol of SELLS called BULLS.
3. Developed a portable implementation of the BF called the Simple Bloom Filter (SBF) in Java. The empirical evaluation of the SBF showed that our designed model of the BF was feasible in the sense that when compared to the analytical, there was a negligible difference.

In general we can conclude that our new space and search efficient data structure for P2P, used by SELLS, is cost effective and space efficient.

5.1 Future Work:

Our future work will seek to:

1. Develop more extensive experiments that study the behavior of the IBF structure used by SELLS given that the BF model showed promising results. Specifically, it is interesting to experiment with executing searches using real P2P file names over the complete IBF structure.
2. Improve our new model for space and search efficient data structure in other networks.
3. Study the implementation and use of BF in wireless sensor networks.

References:

[1] Perera, G., Christensen, K., and Roginsky, A. “Broadcast Updates with Local Lookup Search (BULLS): A New Peer-to-Peer Protocol,” *Proceedings of the ACM Southeast Conference*, (2006), 22(6):124–129.

[2] Fabiano C. Botelho and Nivio Ziviani. “External Perfect Hashing for Very Large Key Sets”

[3] Andrei B., and Michael M.“Network Applications of Bloom Filters”:,A Survey

[4] N. Alon, M. Dietzfelbinger, P. B. Miltersen,E. Petrank, and G. Tardos. Linear hash functions. J. of the ACM, 46(5):667–683, 1999.

[5] Yi Lu, Balaji Prabhakar,and, and Flavio Bonomi. “Perfect Hashing for Network Applications”.

[6] Prosenjit B., Hua. G. J. M, Evangelos K., M. S.,Anil M. Y.T., Pat M. “On The False Positive Rate of Bloom Filters”.

[7] José A., “Adaptado de Database System Concepts - 5th Edition- Indexing and Hashing”

[8] http://www.computerworld.com/s/article/69883/Peer_to_Peer_Network?taxonomyId=016

[9] Graciela P.,Robert K., Anthony W.,“A Space Efficient Local Look-up Search Peer-to-Peer Protocol for Trustworthy Key Distribution”