

A Literature Survey on Bloom Filters

Research Assignment in Computer Science

Parallel and Distributed Systems group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Jelle Roozenburg

8th November 2005



Delft University of Technology

Preface

This document contains a literature survey on Bloom filters and their applications. It is an initial research assignment for my Master of Science project at the Delft University of Technology. The research is done at the Parallel and Distributed Systems Group, Faculty of Electrical Engineering, Mathematics, and Computer Science. The research is part of the I-Share project. I-Share is researching the sharing of resources in virtual communities for storage, communications, and processing of multimedia data.

This research assignment is a preparation for my Master of Science project in which the knowledge of this study will be used to implement and test a p2p file-sharing network using Bloom filters.

Jelle Roozenburg

Delft, The Netherlands
8th November 2005

Contents

Preface	iii
1 Introduction	1
2 Bloom filters	3
2.1 The basics of Bloom filters	3
2.2 Extended Bloom filters	6
2.2.1 Counting Bloom filters	6
2.2.2 Spectral Bloom filters	7
2.2.3 Bloomier filter	8
2.2.4 Compressed Bloom filters	9
2.2.5 Distance-Sensitive Bloom filters	10
3 Applications of Bloom filters	11
3.1 Hyphenation	11
3.2 Spell and password checking	11
3.3 Checking for viruses in network packets	12
3.4 Spam control in email	13
3.5 Web caching	14
3.6 Finding metadata	15
3.7 Data location in a p2p network	15
3.8 Efficient p2p keyword searching	17
4 Bloom filters in p2p networks	21
4.1 Finding similarities in p2p networks	21
4.1.1 Content similarity	21
4.1.2 Mutual friend discovery	22
4.2 Security	23
4.2.1 Misusing a Bloom filter	23
4.2.2 Bloom filters with static bit density	24
4.2.3 Two distinct Bloom filters	24
4.2.4 Challenge-response	25

5	Social network analysis	27
5.1	Social networks	27
5.1.1	Different kinds of social networks	27
5.1.2	Friendster	28
5.2	Crawling Friendster	28
5.3	Data analysis	30
5.3.1	Number of friends	30
5.3.2	Number of friends of a friend	31
5.3.3	Clustering coefficient	34
5.4	User exchange with Bloom filters	37
5.4.1	Mutual friend discovery	37
5.4.2	Online peer discovery	39
6	Conclusions and Future Work	41

Chapter 1

Introduction

Peer to peer (p2p) file sharing networks are very popular these days. Millions of people use these networks to share enormous amounts of digital content. Most of this content is copyrighted material that is illegally distributed, like movies and music. This fact makes the p2p file sharing systems very controversial and very popular to millions of people. The fact that 60% of all Internet traffic consisted of p2p file sharing in the end of 2004 [14], and that this percentage is still growing, shows its popularity.

P2p file sharing is however not only about copyright infringement. The theory behind p2p networks shows enough advantages compared to conventional client-server systems to create many legal opportunities for p2p systems in the near future. One valuable property of p2p systems is that they automatically scale with the number of users, in contrast to a central server with requirements that grow with the number of users. Examples of legal applications of p2p networks are the Skype network [22] for world-wide telephony and the distribution of free software through Bittorrent [3].

The difference between p2p networks and centralized solutions is basically that in p2p networks content and knowledge are distributed over the network users. This introduces many research questions. The Parallel and Distributed Systems group is researching these topics by designing various solutions and testing them in real p2p file sharing networks. Currently, new features are added to an existing Bittorrent client to explore new possibilities of file sharing.

The background of one of these features is discussed in this report, namely *Bloom filters*. A Bloom filter is a data structure that can be used for bandwidth efficient communication of datasets. It also enables network users to send sets of content while protecting their privacy, since a Bloom filter can only be read by users that have similar content.

The first part of this report is an overview of the properties and applications of Bloom filters as found in the current literature. In Chapter 2, the properties of Bloom filters are explained and more complex data structures based on Bloom fil-

ters are introduced. In Chapter 3, some applications of Bloom filters are described. The second part of this report gives a more detailed view of how Bloom filters may be applied in p2p networks. In Chapter 4 applications in p2p file sharing networks with a social overlay are discussed. Applications in p2p file sharing include the discovery of mutual friends, online peers and similar content.

To get an insight into digital social networks, the Friendster [10] social network is analyzed in detail in Chapter 5. The number of friends, the number of friends of a friend and the clustering coefficient are studied. These properties are used to calculate the sizes of Bloom filters used for the applications explained in Chapter 4. From these Bloom filter sizes, the bandwidth usage of the applications can be estimated. We can conclude that Bloom filters enable peers to find mutual friends and online peers with very low bandwidth usage.

Chapter 2

Bloom filters

A Bloom filter is a data structure which can store the elements of a set in a space efficient manner, if a small error is allowed when testing for elements in the Bloom filter. In Section 2.1 these basic properties of Bloom filters are described. In the years after the introduction of the Bloom filter, data structures based on the basic filter were presented by different researchers. These structures are described in Section 2.2.

2.1 The basics of Bloom filters

A Bloom filter is a data structure that can be used to represent a set of elements. One can first add elements of a set to the structure. Later on, the structure can be queried for the membership of elements. So the elements themselves are not stored in the Bloom filter, only their membership. Bloom filters were named after Burton H. Bloom, who introduced them in [2]. In this paper he compared the space/time trade-offs of different types of hash-tables. He found that a new type of hash-table, which is now known as a Bloom filter needed less time to reject elements that are not in the table and less space to store these elements. The drawback of this Bloom filter is that a small error probability is introduced when testing if elements are in the filter.

A Bloom filter consists of a bit array of m bits, which are all initially set to 0. Adding the elements of a set $S = \{x_1, x_2, \dots, x_n\}$ of n elements is done as follows. For each element x_i that is added, k different hash functions h_1, \dots, h_k each with a range $\{1, \dots, m\}$ are used to calculate k different hash values $h_1(x_i), \dots, h_k(x_i)$. We assume that these hash functions map each element to a random number uniform over their range. Then, the bits $h_j(x_i)$ are set to 1, for $j = 1, 2, \dots, k$. The bits in the Bloom filter can be set to 1 multiple times, but only the first time this has effect. The addition of the elements is show in Figure 2.1.

When we want to check if a certain element is in our Bloom filter, a similar approach is used as during the addition of elements. The same k hash functions are calculated over the element y . Then we test if the bits $h_i(y)$ for $i = 1, \dots, k$ are

equal to 1. If one or more of these bits are still 0, the element is certainly not in the set. If all bits are 1, the element was probably in the set, although there is a small probability that the tested bits were set to 1 due to the addition of different elements. Then we have a false positive.

There is a trade-off between the probability of false positives and the size of the Bloom filter. The false-positive probability can be calculated from m and k in the following way.

The probability p of one of the m bits still being zero after the addition of n elements is:

$$p = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}. \quad (2.1)$$

The probability of a false positive f is then equal to the probability that all the k bits that we test are equal to 1, which is equal to

$$f = (1 - p)^k = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k. \quad (2.2)$$

By taking the derivative of Equation 2.2, from simple calculations it follows that for a given m and n , the value of k that minimizes the false-positive probability f is equal to

$$k = \frac{m}{n} \ln 2, \quad (2.3)$$

which gives a false-positive probability of

$$f = \left(\frac{1}{2^{\ln 2}}\right)^{m/n} \approx 0.62^{m/n}, \quad (2.4)$$

and the probability p of one of the bits still being 0 equal to

$$p = e^{-kn/m} = \frac{1}{2}. \quad (2.5)$$

The optimal values for k calculated with Equation 2.3 are not natural numbers. So the optimum has to be rounded to a natural number to get a useful k value. It is

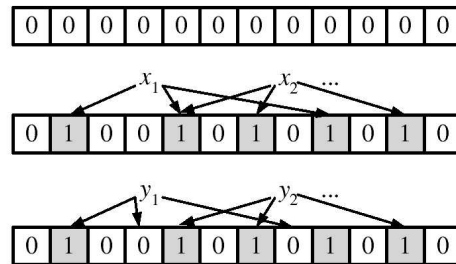


Figure 2.1: The bit array of a Bloom filter. Top to bottom: an empty Bloom filter, adding elements and checking the membership of elements (taken from [4]).

most practical to make k the natural number smaller than the optimal value, so that less calculation is needed to add and test elements in the Bloom filter.

In Table 2.1 are some examples of error probabilities and the sizes of the related Bloom filters.

bits/element	m/n	2	8	16	24
number of hash functions	k	1.39	5.55	11.1	16.6
false-positive probability	f	0.393	0.0216	$4.59 \cdot 10^{-4}$	$9.84 \cdot 10^{-6}$

Table 2.1: The false-positive probability for Bloom filters with different sizes and an optimal number of hash functions.

For large k , using k hash functions demands much computation. In [19] M. Mitzenmacher analyzes the use of linear combinations of two hash functions h_1 and h_2 to create k new hash-functions. The error-probability f does not increase by this.

Example

The space efficiency of Bloom filters is especially interesting when they have to be transmitted over a network.

As an example imagine that two applications (A_1 and A_2) on different computers each have a list of files stored and both want to know the intersection of their sets of files. Let the file list of A_1 be set S_1 with 3000 files and the file list of A_2 be set S_2 with 2000 files. The number of files that they both have is $S_1 \cap S_2$ which contains 500 files. In the communication, file identifiers of 256 bits are used.

The non Bloom filter solution (shown in Figure 2.2a) would be that A_1 sends its complete file list to A_2 . A_2 compares its list with the received list and calculates $S_1 \cap S_2$. Then, A_2 sends back this result to A_1 . The total network-traffic would be $(|S_1| + |S_1 \cap S_2|) \cdot 256$ bits. In this example that would be 112 KByte.

When Bloom filters are used (as shown in Figure 2.2b), this bandwidth use can be decreased. Assume an error probability f of 0.001 is sufficiently accurate. Then Equation 2.4 gives that $m/n = 15$ bits. Both the transmission of sets S_1 and $S_1 \cap S_2$ can be done by a Bloom filter of this size. Hence the new bandwidth usage will be $(|S_1| + |S_1 \cap S_2|) \cdot m/n$ bits, which leads to 6.6 KByte, a reduction of 94%.

The price that is paid for this reduction is the possibility that A_2 concludes that certain files are in S_1 and thus in $S_1 \cap S_2$ which are actually not. Also A_1 might find false positives in the received Bloom filter of $S_1 \cap S_2$. If this error is not allowable, the set $S_1 \cap S_2$ can be sent in a normal list format, so that a Bloom filter is only used for the transmission of set S_1 .

The calculation that has to be done for this operation is more efficient for the Bloom filter solution, because lookups in the Bloom filter can be done in $O(k)$ time, instead of $O(n)$ time for the normal list solution.

2.2 Extended Bloom filters

Bloom filters can be used as a basis for more complex data structures. In this section the Counting Bloom filter, Spectral Bloom filter, Bloomier filter, Compressed Bloom filter and Distance-sensitive Bloom filter are presented.

2.2.1 Counting Bloom filters

In Section 2.1 we have described how a set S can be represented by a Bloom filter $F(S)$. One by one, the elements of S are added to $F(S)$, by setting bits of the bit array to 1. When the set S changes over time, and more elements need to be added to $F(S)$, this can simply be done in the same manner. The deletion of elements from $F(S)$ is however not possible. If element y has to be deleted, bit locations $h_i(y)$ for $i = 1, \dots, k$ can be calculated, but the bits at these locations cannot be simple set to 0, because they might have been set to 1 by the addition of other elements than y . This leads to the impossibility of deletion from the standard Bloom filter.

A solution for this problem was found by Fan, Cao and Almeida [9]. They have designed an efficient protocol for web proxies, in which the proxy servers share their web caches. For this to be possible, the proxy servers send *summary caches* to each other. A summary cache is basically a Bloom filter describing the set of URLs a proxy server has cached at a certain moment. Because URLs also may

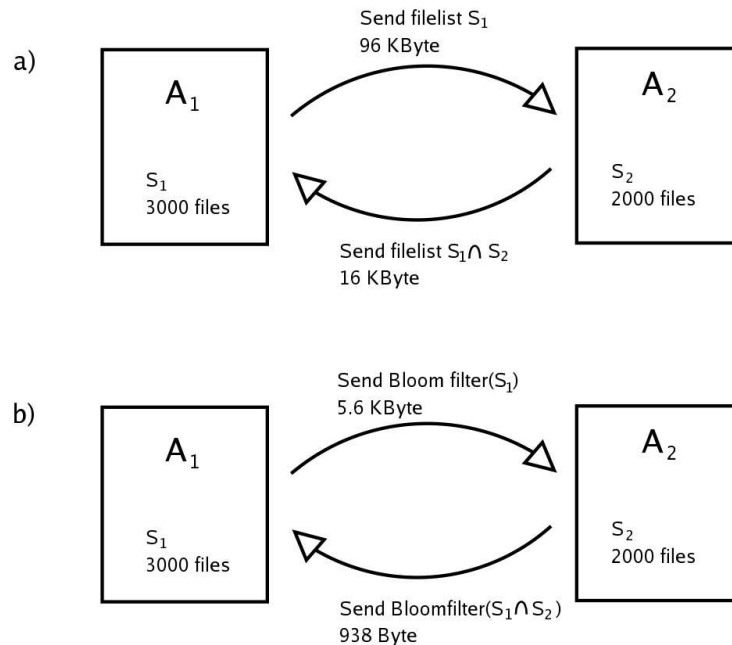


Figure 2.2: The calculation of the intersection of two file lists, by the exchange of (a) raw file lists or (b) Bloom filters of the lists.

have to be deleted from the Bloom filters (for instance, when cached URLs are out-dated), they propose to use a *Counting Bloom filter*.

In a Counting Bloom filter, the bit array of m bits is replaced by an array of m counters of b bits, C_i for $i, 1 \dots, m$. On addition or deletion of an element y , the counters $C_{h_i(y)}$ for $i = 1, \dots, k$ are either increased or decreased by 1. A problem arises when a counter overflows. The proposed solution is to leave the counter at its maximum value, which creates a tiny possibility for false negatives when the counter is decreased down to 0 afterward.

The size b of the counters has to be large enough to guarantee a small probability of counter overflow. For a Counting Bloom filter with m counters, n inserted elements, and $k \leq m/n \ln 2$ hash functions, Fan and others calculated that the probability that the counter with the maximum value $\max(C)$ is higher of equal to a certain $i \in \mathbb{N}$, is equal to:

$$P(\max(C) \geq i) \leq m \left(\frac{e \cdot \ln 2}{i} \right)^i. \quad (2.6)$$

They conclude that, when using 4-bits counters with $i = 16$, yields a counter-overflow probability $P(\max(C) \geq 16) = 1.37 \cdot 10^{-15} \cdot m$, which is sufficiently small for most applications. False negatives are even less probable, because they require a specific chain of additions and deletions.

2.2.2 Spectral Bloom filters

A Spectral Bloom filter [6, 8] is a data structure related to the Counting Bloom filter, which is designed for encoding multisets. It is spectral in the sense that it allows filtering of elements whose multiplicities are within a specific range (spectrum). For a multiset S of n distinct elements from U with multiplicities $\{f_x : x \in S\}$, a Spectral Bloom filter of $N + o(N) + O(n)$ bits can be built in $O(N)$ time, where $N = k \sum_{x \in S} \lceil \log f_x \rceil$.

For any given $q \in U$, the Spectral Bloom filter provides in $O(1)$ time an estimate \hat{f}_q for the multiplicity of q , with $\hat{f}_q \geq f_q$. With high probability $\hat{f}_q = f_q$, otherwise an error E_{SBF} has occurred. The simplest estimate for f_q is calculated as follows. Let

$$v_q = [C_{h_1(q)}, C_{h_2(q)}, \dots, C_{h_k(q)}] \quad (2.7)$$

be the sequence of counter values of the counters on the locations pointed to by the hash values $h_1(q), h_2(q), \dots, h_k(q)$. Now let m_q be the minimum value in v_q . This minimum is the best estimate for f_q of all values in v_q .

To decrease E_{SBF} , two optimizations are proposed by Cohen and Matias, *minimal increase* and *recurring minimum*. When minimum increase (MI) is used, not all counter values in v_y are increased when element y is inserted into the Spectral Bloom filter. All values $C_{h_i(q)} \geq m_y$ have been increased during the insertion of elements other than y , so they are not increased during the addition of y until m_y

has caught up with them. Figure 2.3 gives an example of MI. The counter values before and after adding four instances of element y are shown. The disadvantage of MI is that it does not support deletions from the Bloom filter.

Recurring minimum (RM) is a method that uses the fact that $m_y > f_y$ especially when v_y has only a single minimum. For elements with recurring minimum (more than one minimum), the usual estimate $g_y = m_y$ is used. The elements with a single minimum are also stored in a secondary Spectral Bloom filter with smaller error-probability. The two Spectral Bloom filters together have a far smaller E_{SBF} this way.

Finally, a space-efficient structure is proposed to store the counters of the Spectral Bloom filter, in which counters have different sizes (just enough to store their individual counting values), but still can be accessed in an efficient way.

2.2.3 Bloomier filter

A Bloom filter only allows membership queries on the elements of the stored set S . It is then used to store the membership function $f : S \rightarrow [0, 1]$. Chazelle and others proposed the Bloomier filter [5], a data structure based on Bloom filters that can encode arbitrary functions while maintaining economical use of storage. That is, a Bloomier filter can associate values with the keys stored in the Bloom filter so that it is possible to query for values.

For example, one can store a set of objects S as keys in a Bloomier filter and have one of the color values $\{red, blue, green, white\}$ associated with each object. By querying the objects, the related color is returned. In this case the function encoded in the Bloomier filter is f from D to $R = \{\perp, red, blue, green, white\}$, such that $f(x) = \perp$ for all x outside the fixed subset $S \subseteq D$ of size n .

The possibility of false positives in regular Bloom filters has the following consequences for Bloomier filters and the encoded function f : Queries for $f(x)$ will be always correct when $x \in S$ (no false negatives) and almost always correct when $x \in D \setminus S$ (false positives).

Bloomier filters are implemented as a cascading pipeline of Bloom filters. For illustrative purposes, we take $R = \{\perp, 1, 2\}$. Let A (resp. B) be the subset of S mapping to 1 (resp. 2). The key that is queried is run through a sequence of Bloom

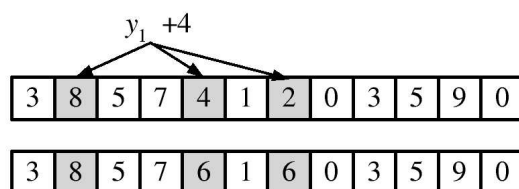


Figure 2.3: The values of the counters in a Counting Bloom filter before and after adding four instances of y . Using minimum increase (MI) only the lower values are increased (taken from [4]).

filter pairs $(F(A_i), F(B_i))$, for $i = 0, 1, \dots, \alpha$. The first pair of the sequence corresponds to the assignment $A_0 = A$ and $B_0 = B$. So a key y that passes the test for membership in $F(A_0)$ and not $F(B_0)$ is probably in A and vice versa. Keys that fail both membership tests are surely not in A nor in B . The problem arises when y passes membership test for $F(A_0)$ and $F(B_0)$. In this case, in one of these Bloom filters a false positive has occurred. The remedy is to fill the next pair of Bloom filters in the sequence with the false positives of the previous stage. We define A_i to be the set of keys in A_{i-1} that pass the test in $F(B_{i-1})$; by symmetry, B_i is the set of keys in B_{i-1} that has passed the test in $F(A_{i-1})$. The number of pairs α has to be chosen such that the sets A_α and B_α are empty. The size of a Bloomier filter with n keys and $R = \{\perp, 1, \dots, 2^r - 1\}$ is $O(nr)$.

2.2.4 Compressed Bloom filters

Because the data structure of a Bloom filter is just a bit array, it can be compressed to make it even more space-efficient. Mitzenmacher researched the possibilities of compression to optimize Bloom filters in [21].

If we consider the bit array of a normal Bloom filter with the optimal number of $k = m/n \ln 2$ hash functions, each bit in the bit array is 1 with probability $p = 1/2$ (as shown in Equation 2.5). The bit array appears to be a random string of m 0's and 1's with each each entry being 0 or 1 more or less independently with probability 1/2. The theoretically possible compressed size z of a bit array is equal to $H(p) \cdot m$, with the entropy function

$$H(p) = -p \log_2 p - (1 - p) \log_2 (1 - p). \quad (2.8)$$

For $p = 1/2$ the length of the compressed bit-string is equal to the length of the uncompressed one. Compression does not gain anything, because with this p the bit string is a sequence of independent boolean variables. Hence we have to choose the value of k to optimize the compressed Bloom filter size z . Mitzenmacher chooses to minimize the false-positive probability f while remaining constant z . Therefore he finds from Equation 2.2 that in a compressed Bloom filter, f is equal to

$$f = (1 - p)^{-\frac{z \ln p}{n H(p)}} \quad (2.9)$$

This function appears to have a maximum when $p = 1/2$ or equivalently, when $k = m/n \ln 2$. Thus the number of hash functions k that is optimal for normal Bloom filters, actually is the worst choice for compressed Bloom filters. In Figure 2.4 and Table 2.2, f is shown as a function of k for a compressed Bloom filter with $z/n \leq 8$ bits. From the figure it is clear that compressed Bloom filters are at least as good as regular Bloom filters.

Compressing Bloom filters has also some disadvantages. First of all, the size of the uncompressed Bloom filter, m , increases for smaller values of k . This increases the memory use when operating on the Bloom filter. Compression obviously costs computation, each filter has to be compressed and later decompressed. Finally

compression doesn't give a guaranteed size of the Bloom filter, only average sizes can be calculated. In practice, when transmitting a compressed Bloom filter, it might become just too big for a network package. Furthermore, using the entropy function calculates theoretical compression ratio; practical compression schemes cannot realize this.

Array bits per element	m/n	8	14	92
Transmission bits per element	z/n	8	7.923	7.923
Hash function	k	6	2	1
False positive rate(f)	f	0.0216	0.0177	0.0108
% less f compared to no compr.		0%	-18%	-50%

Table 2.2: Different false positive rates for at most eight bits per item (taken from [21]).

2.2.5 Distance-Sensitive Bloom filters

Recently Mitzenmacher introduced the distance-sensitive Bloom filter [20], which can answer queries of the form “Is x close to an element of set S or does it have a certain distance from all elements of set S ”, where closeness is measured under a suitable metric. This data structure is implemented using locality-sensitive hash functions.

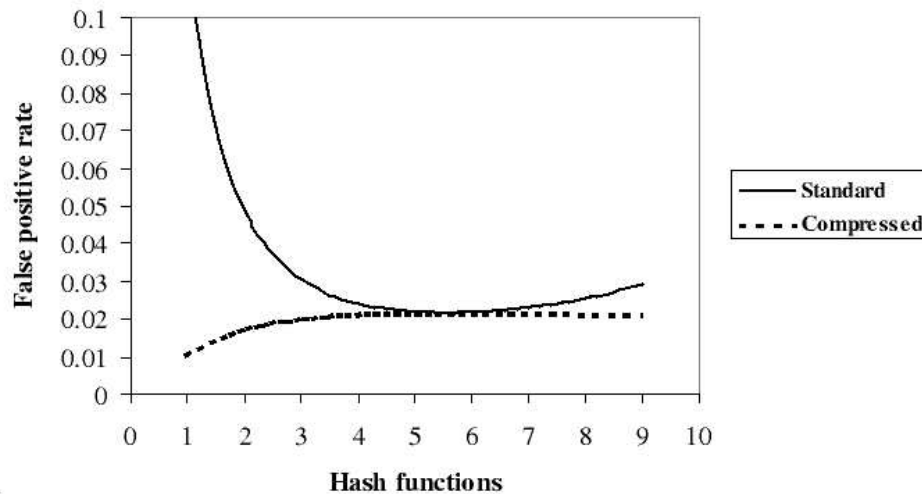


Figure 2.4: The false positive rate as a function of the number of hash functions for compressed and standard Bloom filters using 8 bits per element (taken from [21]).

Chapter 3

Applications of Bloom filters

Although Bloom filters have existed for a few decades now, their applications are not widespread. In the first years after the introduction by B.H. Bloom, the filter was used for the storage of sets of words. Currently, the filter is also used in network applications. In this chapter some applications of (extended) Bloom filters are presented. These applications can be divided into storage of words from natural languages (Sections 3.1 and 3.2), security and spam control (Sections 3.3 and 3.4), web and metadata caching (Sections 3.5 and 3.6) and applications in p2p networks (Sections 3.7 and 3.8). More about applications of Bloom filters can be found in [4]. In Chapter 4 the use of Bloom filters in a social overlay network for file sharing is described.

3.1 Hyphenation

In the article of B.H. Bloom [2] his filter is used for an automatic hyphenation application. The example shows that by using a Bloom filter, the primary lookup table is small enough to fit in the (in 1970 not so big) main memory, and the frequency of disk accesses can be reduced. For this to work, the set containing all English words with irregular hyphenation are stored in a Bloom filter. It is assumed that a total of 500,000 words have to be hyphenated of which 450,000 can be processed by regular rules. The other 50,000 are stored in the filter ($n = 50,000$). Different values for the false-positive probability f are considered, leading to different sizes of the filter m , from 72 Kbit up to 500 Kbit. Occurrences of false positives are filtered out by a secondary, slower lookup. This way, Bloom uses his filter together with a regular hash table as a smaller and more efficient data structure with no errors.

3.2 Spell and password checking

When memories were much smaller than today, Bloom filters were also used in UNIX spell checkers [16]. By putting a dictionary list in main memory the number of slow hard disk accesses could be reduced. One of the solutions for compressing

the dictionary was using a Bloom filter. The disadvantage is that false positives leads to erroneous words being accepted.

When users are allowed to create passwords themselves, it can be a security weakness that they often choose frequently used password strings like existing words. A password checker can solve this problem by disallowing users to use these phrases. It works with a dictionary file in the form of a Bloom filter which stores these easy guessable words. Because users tend to change only one character to work around these kind of checks, Manber and Wu have come up with way to check if there are dictionary words that differ one character from the password [15]. This not done by creating a distance sensitive Bloom filter, but by simply creating a set of all strings with distance 1 of the password and checking them one by one.

In these sorts of password checkers, the Bloom filter has to contain all most used words from the applicable language ($n \approx 50,000$). The false-positive rate can be high, because it will only now and then disallow valid words, which is not a problem. Thus, a relative small bits per elements can be chosen for the Bloom filter. For instance one can choose m/n equal to 8 bits, which gives a filter of 400 Kbit and false-positive rate f of 2.2%.

3.3 Checking for viruses in network packets

Viruses and worms are becoming a more serious problem now that individuals and companies depend more on the Internet and digital data. The scanning of viruses is basically the comparison of file/network data with a stored list of suspicious strings. Dharmapurikar and others [7] proposed to create a high speed embedded Bloom filter to scan network packets in real-time. They designed a sort of firewall that uses deep packet inspection to search for viruses. To be able to scan packets in real-time (2.5 Gbps) for 10,000 virus strings, they use Bloom filters embedded in hardware (FPGA). The calculation of the k hash functions is done in parallel and false positives are filtered out by a secondary test module (see Figure 3.1). They conclude that a Bloom filter is a data structure that can be embedded efficiently.

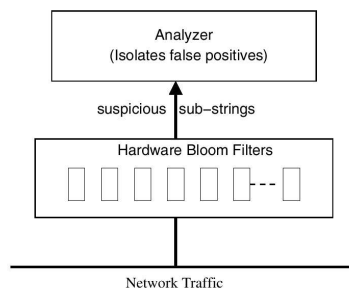


Figure 3.1: Bloom filters scanning all traffic of a gigabit network for predefined signatures (taken from [7]).

In the prototype implementation they used 7 parallel Bloom filters each having a size m equal to 20480 bits to store 1434 virus strings, which gives an f equal to $1/2^{10}$. Together the Bloom filters store 10,038 strings.

3.4 Spam control in email

Bloom filters are also used against spam. The idea of the Loaf system [11] is that email from people you don't know is possibly spam and is initially blocked. After approving it, the sender will be added to a white list. If you exchange these white lists with your trusted friends, there is a reasonable chance that after a while a trustworthy white list is created with all the members of your community. Now anyone can distinguish complete strangers, people known by your contacts and your own contacts.

All white lists with email-addresses are stored in Bloom filters. These filters are small enough to attach to your email in a normal MIME format. The Loaf system automatically checks for these attachments and manages the white lists of your contacts.

This system can be attacked in a few ways:

- Ex-girlfriend attack: It's easy to search for specific email addresses in the white lists of your contacts, so your ex-girlfriend could check which girls you're corresponding with. This positive membership-test can however always be a false positive. A trade-off has to be made between accuracy and privacy/size.
- Dictionary attack: In practice, someone can do a brute-force attack on your Bloom filter. This way all your email contacts (and some false positives) can be retrieved. This is the same problem as the ex-girlfriend attack, and increasing the false-positive probability is a solution.
- Fake Bloom filter attack: One could send a fake Bloom filter (for instance, a bit array with only 1s) and pretend to know all email addresses in the world. This could be filtered out by limiting the maximal density of trusted white lists.
- Me-too attack: This is the case when some one sends not his one white list, but one of the lists he received from his contacts. In this case this user can act like having very similar contacts as someone else. This has no direct advantages for the attacker and can be undone simply by seeding all hashes with the email address of the sender. This way, a forwarded Bloom filter is scrambled for every receiver.

The idea of Loaf is that email contacts that joined Loaf are always trusted. When Loaf users are intentionally sending white lists full of spammer email addresses around, the system doesn't work. A solution could be to send around also black

lists of misbehaving Loaf users or to simply ignore these attackers after they have been found to be malicious.

The size of the Bloom filters attached to your emails are not explicitly defined by the Loaf creators. An example is shown using a Bloom filter with a size m equal to 4314 bits and 10 hash functions. Assuming the optimal number of hash functions is chosen, this example stores 622 email addresses. The email attachment consists of the email address of the sender, some comments, a description of all used hash functions (including salts), the size of the filter m and finally the bit array of the filter encoded in UTF-8. In total this attachment is only 1.2 KByte in size, demonstrating the efficiency of Bloom filters for these purposes.

3.5 Web caching

In Section 2.2.1 Counting Bloom filters were explained. These Bloom filters were introduced by Fan and others in their paper about distributed web caching [9]. They propose a new protocol for web cache sharing and compare it to the existing Internet cache protocol (ICP) [26]. Web cache sharing is the cooperation of a group of proxy servers. When a proxy has a cache-miss, it looks for another proxy that most certainly stores the web page related to the requested URL.

ICP has implemented this by sending queries to other proxies in case of a cache-miss. This generates $O(N^2)$ queries and computation for N proxies working together. The new protocol reduces communication by exchanging *summary cache* between the proxies periodically. Summary cache is basically a Counting Bloom filter of URLs stored on the proxy. Besides a Counting Bloom filter, also a regular Bloom filter representation is kept in each proxy, which bits are changed whenever a counter in the Counting Bloom filter drops to zero or is increased from zero. These regular Bloom filters are exchanged between the proxies periodically by sending the difference from the Bloom filter that was sent previously. Hence, all proxies know the URLs of their group mates without much overhead and are able to contact the appropriate proxy without querying all $N - 1$ of them.

In the experimental setup, the proxies are assumed to store about 1 million web pages of 8 KiloByte in their 8 GigaByte memories. The size of the Bloom filters equals 16 bits per URL. In the Counting Bloom filter, 16 counters of 4 bits are kept per URL. This leads to a false-positive rate f equal to 0.00048. In a group of N proxies, each proxy has to store a Counting Bloom filter of 8 MegaByte and N regular Bloom filters of 2 MegaByte.

False positives lead in this case to querying URLs that are not available at the requested proxy. The bandwidth used for this is much lower than the reduction established by the Bloom filter exchange. Simulations of the protocol show a reduction of 50% of the bandwidth and 30 to 95% CPU usage in comparison to ICP. If there is more bandwidth available than memory, one can choose smaller Bloom filters, resulting in more false positives and inter-proxy communication.

3.6 Finding metadata

When a cluster of metadata-servers is used to store metadata of a file storage, the main problem is to find out which metadata server is holding the data of which file. In [27] a comparison is made between two solutions for this problem. One using Pure Bloom filter Arrays (PBA) and the other using Hierarchical Bloom filter Arrays (HBA).

A PBA is an array of regular Bloom filters. Every Bloom filter stores the set of files that have their metadata on one of the p servers. When a lookup for a file is done, this file is tested against all Bloom filters in the PBA. A hit is defined by a situation in which the Bloom filter of the server that has the file's metadata passes the test and all other Bloom filters don't. Otherwise, a miss has occurred. Because each of the p Bloom filters can give a false positive with a probability defined by Equation 2.4, the probability of a hit is equal to

$$P(\text{hit}) = (1 - f)^p \approx (1 - (0.62)^{m/n})^p. \quad (3.1)$$

In a realistic situation, they conclude that storing the data in Bloom filters with reasonable accuracy still demands too much memory on all metadata-servers. For this conclusion they assume that 200 metadata servers each store 500 million files using 16 bits/file to maintain a $P(\text{hit})$ equal to 90%. They propose the application of HBAs to reduce the memory usage of the system.

The idea of HBAs is that a relative big percentage of the queries are done over a relatively small part of the files. In an HBA a hierarchy is implemented in which popular files are stored with more bits/element (more m/n) than less popular files. This is done using two PBAs, with different accuracy for files with different popularity. A Least-Recently-Used list is kept to find out which files are stored in the Bloom filters and which in the second level Bloom filters. This way, a better hit ratio can be realized with around 50% less memory usage.

3.7 Data location in a p2p network

Bloom filters are used for locating data in the OceanStore p2p network [12]. Locating data is the process of finding a document identified by its unique identifier. Because documents are replicated on multiple peers, the nearest replica should be found.

In [25], Rhea proposes a hybrid location algorithm that combines the advantages of both deterministic and probabilistic routing. It appears that if a replica is far away from the query source, deterministic location is nearly optimal, but in case of replicas nearby the query source, there is far more overhead than with probabilistic location. The hybrid location mechanism starts by using probabilistic location to search if replicas exist less than d hops away. If they are not found, the deterministic location takes over.

Because the probabilistic location mechanism uses Bloom filters, we will explain it in detail. The mechanism consists of two algorithms, the query algorithm, which routes queries from node to node in search of a replica, and the update algorithm, which propagates location information if the set of content is changed on a peer. This location information on each peer is stored in a collection of Attenuated Bloom filters (ABF).

An ABF is an array of d regular Bloom filters F_1, \dots, F_d . In each Bloom filter F_i , the document identifiers are stored of the documents that can be found after i hops over the network. This way, all documents that can be found within d hops are stored in one ABF. In the overlay network of the p2p system, every peer has a number of neighbors. For each neighbor a separate ABF is kept with the documents that can be found via that peer. For instance, some peer has a neighbor n_1 and stores the related ABF A_1 . In this A_1 , F_1 stores the documents that reside on n_1 itself, documents in F_2 are all documents that are stored on a neighbor of n_1 , etc. An example of a simple network with an ABF is shown in Figure 3.2.

With this collection of ABFs as location information, it is possible to forward a query to the neighbor with the highest probability of having the nearest occurrence of the searched document. This is what the query-algorithm does. False positives in any of the Bloom filters can lead to the forwarding of queries to peers that do not store the document. If this happens, the search is continued with deterministic routing. In every query, a list of at most d peers that have already forwarded it are stored to be sure it is not forwarded in a loop.

The update algorithm has the function of updating all ABFs when new documents are added to a peer. This is done by letting each peer not only store an ABF for each outgoing link, but also a copy of its neighbor's view of the reverse direction. When a new document is added to a peer, it calculates the changed bits in its own

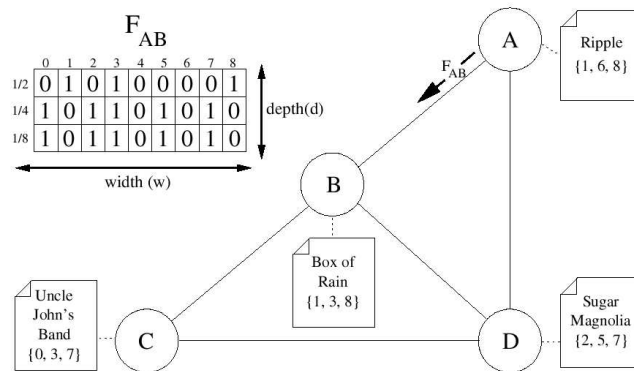


Figure 3.2: An example network with the ABF for the outgoing link AB . In the Bloom filter on row i of the ABF the documents are stored that are i hops away on the network (taken from [25]).

filter and in each of the filters that represent its neighbors view of this peer. These bits are then sent to its neighbors using a sort of differential compression, so that their view of the changed peer is kept up-to-date. This way, changes propagate over the network. To stop this propagation to flood the network and to overfill the Bloom filters in each ABF, different sorts of filtering are proposed.

Using the same scheme, also routing over networks can be implemented in which packets are forwarded along the network to the node holding a Bloom filter to the location the packet has to go to.

The size of the ABFs depends on the average number of neighbors each peer has and the average number of documents stored on each peer. The choice for a certain false-positive rate f is not stated in the paper. Assume that every peer has approximately 10 neighbors and 10 documents. Furthermore, d is chosen equal to 4 and a Bloom filter size m equal to 10,000 bits. Then, each peer would store 20 ABFs with a size of 40,000 bits. The false-positive rate of F_i with $i = 1, \dots, d$ in the ABFs would be approximately equal to $(0.62)^{10^{4-i}}$.

3.8 Efficient p2p keyword searching

Reynolds and Vahdat show that Bloom filters can be used in distributed keyword search [24] for p2p systems. They propose a distributed hash table (DHT) solution for multiple keyword searching in distributed storage.

The search process is done in three steps. First, the search string is divided into separate keywords k_1, \dots, k_n , which are looked up in a DHT. The search for keyword k_i results in a peer-address p_i , that stores the document-identifiers related to k_i . Second, the peers p_i for $i = 1, \dots, n$ are queried for their resulting sets of document identifiers D_i . These are combined by taking the intersection $D = D_1 \cap \dots \cap D_n$ of these sets. This leads to a search method in which all keywords have to be present (AND-search). Finally, the documents in D are retrieved by using a second DHT, that maps document-identifiers to peers on which the documents are stored.

Bloom filters are used in the second step to efficiently calculate the intersection of the sets on different peers without sending raw lists of document identifiers around. Each of the sets D_i usually has many more elements than the final result set D , so sending these as Bloom filters can reduce much bandwidth. Actually, the use of Bloom filters is much like the example in Section 2.1. An example of taking the intersection of documents on two peers is given in Figure 3.3.

The more keywords are used, the more probable it is that D contains some documents included because of false positives. This is because the intersection is implemented stepwise, with Bloom filters used to communicate each temporary intersection result to the next peer. To solve this, one can choose to let p_n send the intersection result D back to p_1 , that will calculate $D \cap D_1$ to remove false positives and sent the result once more along every peer. Because the set D is relatively small, this communication can be done without using Bloom filters. This solution

trades bandwidth for accuracy.

To reduce bandwidth, the use of caches is proposed. If p_i knows already D_j , because it has been send raw or in a Bloom filter during a result-intersection process, it can use this information (for some time) to calculate direct or faster search results.

The optimal size of the Bloom filters is calculated in case of two keywords. Let the sets of document identifiers related to those keywords be A and B . If the total bandwidth used for the intersection is minimized, the optimal Bloom filter size can easily be calculated to be equal to

$$m = |A| \cdot \log_{0.6185} \left(2.08 \frac{|A|}{|B|j} \right), \quad (3.2)$$

where j is the size of a file identifier. Some optimal values for m are shown in Figure 3.4.

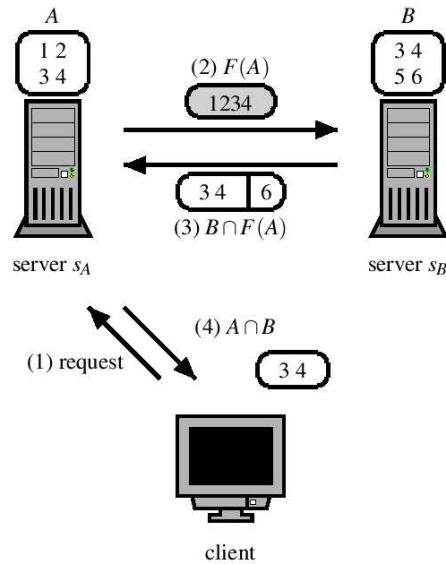


Figure 3.3: Bloom filters help reduce the bandwidth requirement of "AND" queries. The gray box represents the Bloom filter $F(A)$ of the set A . Note the false positive in the set $B \cap F(A)$ that server s_B sends back to server s_A (taken from [24]).

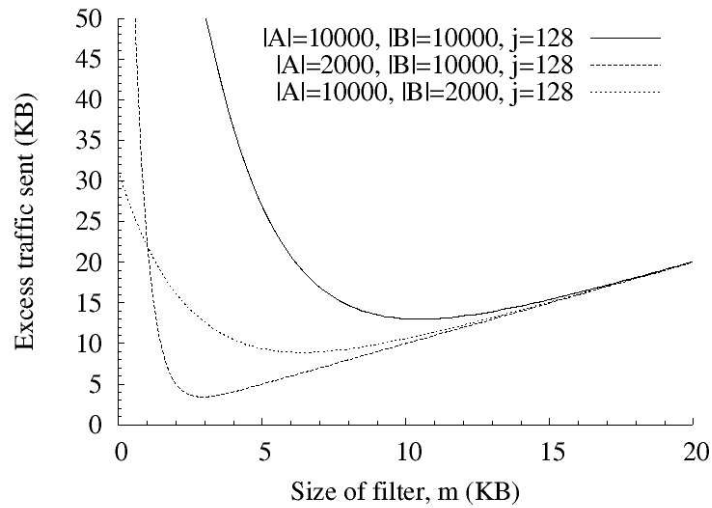


Figure 3.4: The expected bandwidth use as a function of the Bloom filter size m , for different sizes of A and B (taken from [24]).

Chapter 4

Bloom filters in p2p networks

In this chapter we describe how Bloom filters may be used in p2p file-sharing networks. In these networks it is very important to have efficient communication and a sufficient level of privacy and security. By describing two file sharing applications that use Bloom filters, the security and privacy aspects are discussed.

4.1 Finding similarities in p2p networks

In large social p2p networks clusters may emerge. Automatically peers with similarities cluster together, called semantic clustering. This can be similarity in the sense that peers have the same group of friends or the same taste/interests. Clustering is efficient, because peers with the same interests are often searching for the same types of content. These searches can then be started or even limited to their own cluster. For this reason it is interesting for peers to find other peers with similarities.

Communication between two peers can be made very efficient if Bloom filters are used in both directions to represent the datasets. This can be done if the communicated sets are subsets of the sets that the peers own. When peers try to find similarities this is the case, so that an efficient Bloom filter approach is possible.

Below the search for peers with similar friends and with similar content are explained.

4.1.1 Content similarity

To identify peers with similar content taste, a peer can measure the similarity of its shared content with other peers periodically. This can be done with Bloom filters without literally presenting the content to the other peer. This enables peers to get a notion of similarity while protecting privacy. The advantage of Bloom filters is that by allowing a relatively big false-positive probability, both the privacy is protected and the filter size is kept small.

When a peer initiates a content similarity measurement, it sends a small Bloom filter of all its content to the peer to be measured. This peer can make the decision if it is interested based on the number of its files it finds in the receive filter. When no similarity is detected or when the peer doesn't want to cooperate, it replies with a 'no similarity' response. Otherwise, it can send a similar Bloom filter back storing its contents.

One can also choose to create a two step protocol. In the first step peers decide if there is any similarity while using very limited bandwidth and protecting their privacy. If similarity is discovered, in the second communication step more detailed content information is exchanged.

4.1.2 Mutual friend discovery

Instead of content information, peers can also exchange Bloom filters storing the permanent identifiers of peers that are their friends. This way, peers can measure to what extent they have a similar social neighborhood. This measurement can be used as part of a trust function: if a peer shares many friends with me, it must be a respected member of my community.

Mutual-friend information can also be used to find recent ip-address properties of peers. This is important as many peers have new ip addresses or a new port number every time they come online. To solve this, every peer sends *hello*-messages to all peers that are his friends when coming online. Assume peer p_1 comes online and does this. Some of his friends it will reach, because they are both online and have not changed ip or port information after p_1 connected to them previously. Most of them however will be offline or online with new addresses. When p_1 knows all mutual-friend information, it can use that to find these peers through the friends that it can connect to. Of course, it always needs a small portion of friends to be connectible, otherwise it is completely disconnected from the social network. The only option is then to query a super-peer for address information of his friends. Super-peers are always available at static addresses.

The process of retrieving mutual-friend information of friendly peers is equal to finding similar content. A peer broadcasts his friends list stored in a Bloom filter to all of its friends. These respond with a Bloom filter of the subset of the list with mutual friends.

When the mutual-friend information is retrieved, peers can be connected to ask for the address of a certain friendly peer. One can also send a Bloom filter storing peer identifiers of all peers that one does not know the current status of. The receiving peer can then respond with a list of the addresses of the requested peers that it knows of. This we will call online peer discovery.

4.2 Security

In the applications above Bloom filters are a very effective data structure. With their small size they make it possible to exchange them using little bandwidth. In this section, some solutions are proposed to gain both from the compactness of the Bloom filter, but still be able to ensure that the creator has constructed it truthfully.

4.2.1 Misusing a Bloom filter

In the p2p applications described in Section 4.1, a peer sends a Bloom filter to prove that he has a certain collection of knowledge. This can either be content or information about other peers (such as their permanent identifiers). The receiver peer trusts this information and uses it to decide if it is interesting to exchange information with the sender. The weakness in this system is that it's possible for the sender to create a Bloom filter and change it, to make the receiver believe that its dataset is bigger than in reality. From this, the sender can gain advantage.

In the application of content similarity, the Bloom filter is filled with the set of content (files) of the sender. If the receiver concludes that the sender owns files similar to his, he will exchange information of his other content with him. In this case, tricking the receiver to think that the sender has many files, is in his advantage. This advantage can easily be realized by setting all bits in the bit array of the Bloom filter to 1, which will makes the receiver think that the sender has indeed all possible files. Many peers will conclude this peer is very interesting and will for instance route future searched over this peer, thereby giving him more influence. Only when peers are requesting actual files from the malicious peer, they will find out that these files are not there or not the files they expect. Without any security, it would be easy for a malicious peer to misuse the Bloom filter communication this way and gain influence over the p2p network.

Similar alterations to Bloom filters used for communication in the other applications give peers the possibility to pretend they know all peer identifiers or share all friends with some one. Also this misuse makes it possible for peers to increase their prestige in the network and thereby reducing network quality.

These easy hacks should be made impossible for peers. That's why a receiving peer should be able to check if the sender really owns the set stored in the Bloom filter sent by it. The receiver will be especially interested in those elements for which it tests membership on the Bloom filter and on which it bases its decisions. Most often, this will just be a subset of the data that was stored in the Bloom filter.

So for the elements that pass the membership test of the receiver peer, this peer wants to check if the sender really owns them. If the other bits in the Bloom filter are legitimate is not interesting for the receiver.

4.2.2 Bloom filters with static bit density

A first step in preventing Bloom filter alteration is to agree upon a Bloom filter with standard properties that has to be used by all peers.

One can choose a static size m of each Bloom filter, but this will give a false-positive probability dependent on the number of elements in the Bloom filter. This way, the more elements are added to a filter, the more errors will occur and the harder it is to check if these elements were really owned by the sender. In theory, a malicious peer could send a bit array only consisting of 1s and pretend it owns all elements that exist.

A better solution is to agree upon a static false-positive probability. This yields in a static number of bits per element. Bloom filters will then grow linear with the number of elements in it. The advantage is that Bloom filters will always be optimal in their size, but tricks like the bitwise ORing of Bloom filters to compare them can only be done with filters with similar size and (number of) hash functions. In these Bloom filters the probability that one of the bits in the bit array is 1 is approximately equal to 0.5 (see Equation 2.5). The maximum number of bits that has been set to 1 after adding n elements, using k hash function is $n \cdot k$. Using an optimal value for k this yields to a maximum of $\ln 2 \cdot m$ 1s in the bit array.

These facts about optimal Bloom filters disable malicious peers to set many random bits of the bit array to 1. Receiving peers can easily check if the percentage of 1s in the Bloom filter exceeds $\ln 2$ and, if so, ignore the filter. The only way to pretend to have more elements is to send a longer filter. This costs the malicious peers more bandwidth and only enables them to pretend to own a limited number of random elements.

4.2.3 Two distinct Bloom filters

If more security is demanded, one can choose to let all sets be communicated by sending two distinct Bloom filters. Both filters will have an equal and static false-positive probability as above, but different hash functions are used.

When the receiving peer has concluded that an element is in the primary Bloom filter, it will check if this element is also a member of the secondary filter. If it is, then this element must be owned by the sender, if it is not, then the receiver will conclude that he has either found a false positive in the primary filter or the sender is a fraud. Because the false-positive probability of both Bloom filters are known to the receiver, he can find out if errors are due to false positives or if he has to flag the sender as malicious.

It is impossible for the sender to create the Bloom filters in such a way that they store similar elements when they are created from random placed bits. Assume that the sender owns 100 elements, but want the receiver to believe he owns more. Therefore he creates two Bloom filters of 1000 times the static bits per element. After adding his real elements, he sets random bits to 1 so that finally half of the bits of both Bloom filters are set.

What he actually did, is creating two Bloom filters with 100 mutual elements and in each filter approximately 900 random elements. The probability that each of the elements owned by the receiver equals one of the random elements in the primary Bloom filter is equal to the false-positive rate f_1 . The probability that this element is a member of the secondary Bloom filter is equal to the secondary false-positive rate f_2 . This leads to a total probability that the sender can pretend that it has an element of the receiver equal to $f_1 \cdot f_2$. This is also the false-positive probability of the two Bloom filters together.

Instead of sending one Bloom filter of m/n bits per element, one can better send two Bloom filters with each a size of $m/2n$ bits per element. The total size of this solution equals the size of sending a single Bloom filter and it is very hard to create two Bloom filters that represent the same elements by setting random bits to 1.

4.2.4 Challenge-response

The solutions above make it hard for a peer to pretend that it owns certain elements. In practice, these elements can be file identifiers or user identifiers. If a receiving peer not only wants to be sure that the sender owns a file identifier but also the file itself, it has to challenge the sender to prove that it owns the file as well. This communication, where the receiver demands the sender to respond with some private information, is called a challenge-response.

A challenge-response in which the sender has to prove it owns some file, can be executed as follows:

- The challenger asks for certain randomly selected parts of the file and the responder sends these back.
- The challenger sends a random *salt* to the responder, who rehashes the file with this salt and sends back the hash value.
- The challenger sends a list of *salts* to the responder, who uses these to create a new Bloom filter of all his files and sends this back

A salt is an initialization vector to use as an input of a secure hash function. The salt will influence the hash value of all objects, so that it is only possible to find a hash value if one knows both the salt and the object itself. Because the responder does not know with with salt it has to hash the file, it cannot do it in advance.

The advantage of the third option, is that the challenger receives prove of all the files in the Bloom filter at once. This is however computationally intensive, because very big files might have to be rehashed. The first option does not require this hashing of files, but gives no guarantee that the sender owns the whole file.

The problem of these options is that the challenger has to test the responses it receives. So it has to either own a copy of the file itself or know someone it trusts that does. Otherwise the challenge-response is no use.

Chapter 5

Social network analysis

In Section 4.1 some useful applications of Bloom filters in p2p file-sharing networks have been mentioned. The file-sharing networks that will be subject of our research use a social overlay network. In the near future we want to research how Bloom filters can be used effectively and securely in social p2p networks. For this reason digital social networks are analyzed. In Section 5.1 general properties of social networks are explained. Section 5.2 describes the collection of friendship information of one such network, Friendster, using a Web-crawler. This information is analyzed in Section 5.3. Although it is not sure that data of this network is representable for p2p file-sharing networks in the future, it is a valuable first order estimate. In Section 5.4 the data collected from Friendster is used to make an estimate of the size of the Bloom filters to be used for exchanging sets of users.

5.1 Social networks

Recently digital social networks have become more and more popular. Evidently people like to digitize their friendships and use the Internet to meet new people. The Internet enables everybody to scale their group of friends from the people they normally meet to friends worldwide.

5.1.1 Different kinds of social networks

One of the first social networks was initiated by I Seek You (ICQ) in 1996 [17]. This messenger makes it possible to store your list of friends, see who is online and exchange messages or files with them. Soon, all kind of similar messengers were created like MSN Messenger, AOL Instant Messenger, Yahoo! Messenger, Google Talk, .NET Messenger Service and Jabber. The newest versions of these application add many features like playing games, Voice over IP and video conferencing. The communication in the messengers is limited between you and your direct contacts.

Other digital networks try to let users browse the total social network. Is is then

possible to search from friendship to friendship and reach many people. The idea of these networks is based upon the small world phenomenon [18]: a random person in a large highly connected network always can be reached through a short chain of social acquaintances. This leads to digital social networks that enable everyone to reach many people in a few clicks.

These networks also come with many features. They create a detailed profile for every user with their interests and social goals. Users can also communicate in all sorts of ways and join communities. Examples of these networks are Orkut [23] (10+ million members), Friendster [10] (20 million members) and aSmallWorld [1](100,000 members). These social networks can be seen as a large graph, with people as vertices and their friendships as edges.

5.1.2 Friendster

One of the social networks is called Friendster [10]. On this network, anybody can join for free and without invitation. It has around 20 million users worldwide. When subscribing it is possible to add a photo and all kinds of personal information. Then, one can search for people and ask them to be your friend. After a group of friends has been set up, one can exchange messages, play games or join discussion groups. Recently, Friendster added the option to see who has visited your profile.

Because Friendster has many users and a simple structure, we choose it to be analyzed in the next section.

5.2 Crawling Friendster

To collect information from the Friendster network, we have written a crawler in Python. A crawler is an application that automatically parses web pages and retrieves links to other web pages to parse subsequently. This way, the information from a whole network of web pages can be retrieved. In this case, the pages contain friendship relations between members of the Friendster network.

The Friendster network only discloses user information to users who are logged in. So the crawler has to periodically log in and save cookies to prove that it is a valid member. Then the crawler will bootstrap using a random Friendster user and start a breadth first iteration over the social network graph. The crawler stores a list of users that it has already crawled, a list of relations and a queue of people yet to crawl.

The simple algorithm executed by the crawler is:

1. Put the bootstrap user in the queue.
2. Pop a user from the queue, or END if the queue is empty.
3. Parse the web page listing all his friends and add them to the queue if new.

4. Goto 2.

number of crawled users	14,010
number of users known/in queue	649,528
number of crawled users with hidden friendships	2,703
number of directed relations	3,608,981
average number of friends per user	258
average number of unique fofs per user	7524
clustering coefficient of network	0.0320

Table 5.1: Properties of the crawl of Friendster.

Unfortunately, Friendster gives people the option to hide their friendships from every Friendster users and only showing it to their direct friends. This disables the crawler to actively retrieve the friends of these people. Because a friendship is detected from both friends having it, the hidden friendships can be retrieved passively if one of the friends does not use the hide option.

After a continuous crawl time of over 3 days, we have constructed a dataset of around 14,000 users completely crawled, 650,000 known users (in the queue), and 3,600,000 unidirectional friendship relations. The latter means that if a friendship relation is crawled from both sides, the relation is stored twice. All numerical properties of the crawl are shown in Table 5.1 and will be further discussed in Section 5.3.

We did not crawl the complete network, because this would take too much time. A fast calculation shows that with our crawler, the process of crawling all 20 million users of Friendster, would take almost 11 years. This is caused by the fact that we use only one sequential crawler and that the bandwidth to the Friendster website is very low.

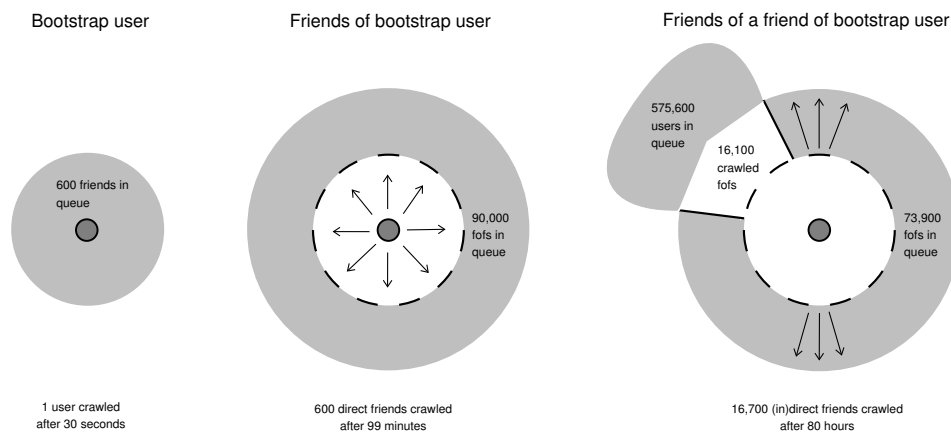


Figure 5.1: Schematic overview of the crawling process.

In Figure 5.1 the crawling process is shown schematically. The crawl is built up around the bootstrap user, who had 600 friends. The second ring, representing fofs of the bootstrap user, contains 89,000 users. The number of users in these rings grows exponentially because each of the users has on average 258 friends of his own. For this reason, only a part of the ring of fofs could be crawled.

The average number of friends of users in the first ring (direct friends of the bootstrap user) and the crawled users in the second ring (fofs of the bootstrap user) are equal to 149 and 37 respectively, if we calculate it from the numbers in Figure 5.1. These numbers are different from the average mentioned before. This is caused by the fact, that the figure omits the fact that the network is clustered. Direct friends of the bootstrap user are probably also friends of each other and fofs in the second ring are likely to have friends in the first or second ring. Also the fact that some users have hidden their friendships explains the different numbers. These users with private friendship data are excluded during the calculation of the average number of users and fofs.

If we calculate the time to crawl one user from the 3 parts of Figure 5.1, we get 30 seconds for the bootstrap user, 9.9 seconds for the next 600 users and 17.2 seconds for the rest of the users. The slow crawl of the bootstrap user can be explained by the need to log in to the Friendster network and the fact that the bootstrap user has many friends. The gradual increase of the crawl time during the crawl is caused by the fact that the administration of users in the user-done list and the queue takes more time when these data structures are bigger.

5.3 Data analysis

The output of the crawler is a large list of friendship relations which form a graph of users. In this section we will analyze this graph to get a notion of the number of friends and the number of unique friends of a friend (fofs) that people have. Furthermore, the clustering in the graph is studied.

5.3.1 Number of friends

The number of friends of each user can be easily determined by counting the number of outgoing friendship relations. In Figure 5.2 the number of friends of all crawled users are depicted, ranked according to the number of friends of a user. It can be seen that a single very social user has 1000 friends. Five crawled users had only one friend. The average number of friends is equal to 258 friends, which seems a lot. Friendster has disabled users to have more than 1000 friends, so this explains why it is also the maximum found in the crawling.

The number of friends is an important property of a social p2p network. Direct friends can be used to find content, exchange interesting peers or communicate who is online. The fact that many users have many friends on the Friendster network is promising for the stability of social p2p networks.

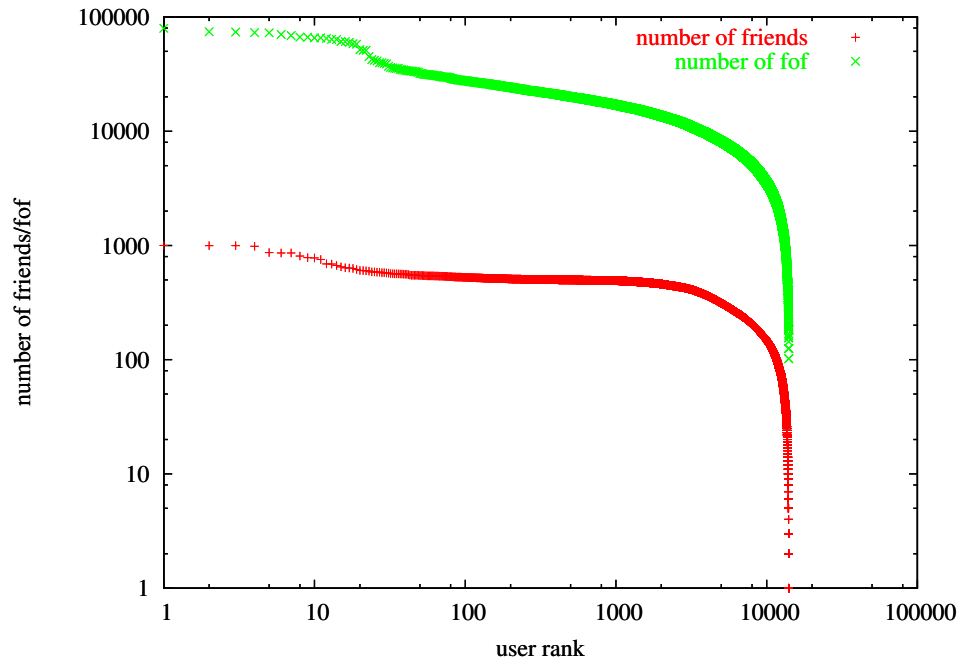


Figure 5.2: The number of friends and the number of fofs per user (independent rankings for friends and fofs).

In Figure 5.3 the probability mass function of the number of friends is plotted. It shows that having around 150 friends is most probable, users with more or fewer friends are harder to find.

Another interesting top is found around 500 friends. It seems that this top is caused by a group of users that is really active in finding friends and finds it a challenge to have around 500 friends. We have found that until recently, 500 was the maximum number of friends a Friendster user could have. Part of the users had the habit to quickly 'collect' friends up to the maximum number. When this happened, they opened a new account. This explains why the probability for a user to have 500 friends is higher than it is to have around 375 friends. Above 600 friends there is a steep decrease in the probability function, almost no one has this many friends. Apparently, the friend-collectors did not have the energy to increase their number of friends to the new maximum.

5.3.2 Number of friends of a friend

Besides the number of direct friends, we have also studied the number of friends of a friend (fofs). For each person in the network, the list of unique fofs can be found by taking the union of all lists of direct friends of his friends. It is important to remove identical friends from the union of friends-lists so that only unique fofs are counted. In Figures 5.2 and 5.3 the number of unique fofs is shown.

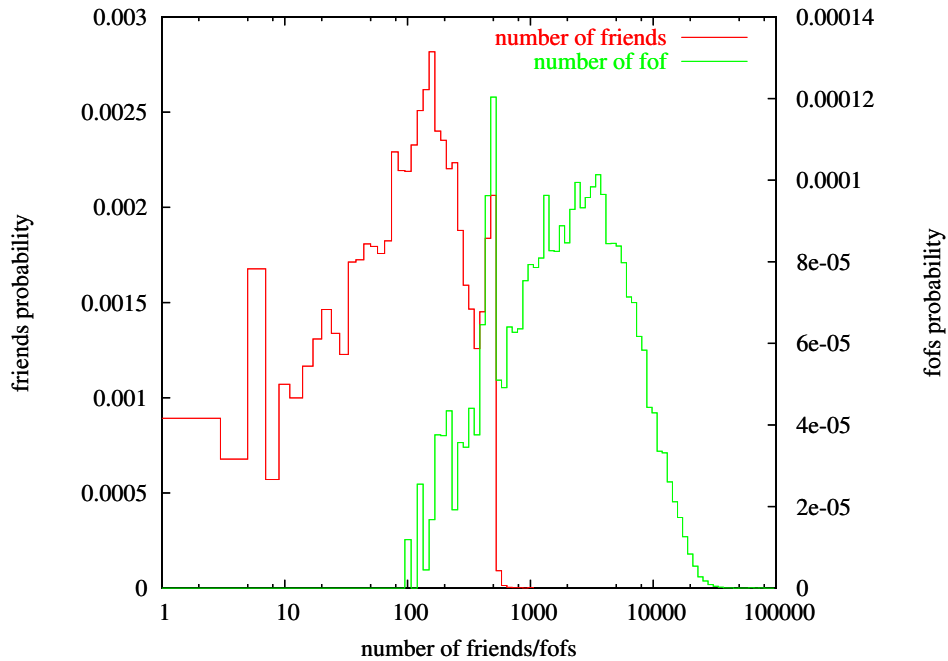


Figure 5.3: The probability mass function of the number of friends and the number of fofs in Friendster.

The figures show that people always have a large number of fofs. This is exactly what the small world phenomenon describes: in only two steps, every user knows a large number of other users. The number of fofs is in the range from 102 to 90,000. It is the bootstrap user who has the maximum number of fofs, because he is in the middle of the crawled graph. The other users are often too close to the sides of the graph to have all their fofs crawled.

Some comments have to be made on these data. As said before, not the complete Friendster network has been crawled and the crawl is executed breadth first. Therefore, the users on the sides of the crawled graph are still in the queue to be crawled and thus have only one friend. These users are not included in the plotted data, but the users with a distance equal to one from the sides of the graph are. So the number of fofs in reality is higher than plotted in the figures. Unfortunately, most users are indeed on the sides of the graph, so that this effect can not be neglected. Only taking into account users with a distance of two from the sides is no solution for this problem. These users are likely to have just a small number of friends, otherwise they had known some friends at the sides of the graph.

The fofs probability mass function shows a bell-curved shape. Only a noticeable peak at around 500 fofs is an exception to this shape. This peak can be explained by the similar peak in the friends probability mass function at the same number of friends. Imagine that many of the direct friends of the bootstrap user have 500 friends themselves. Because of the limited crawl, the only fofs that these users

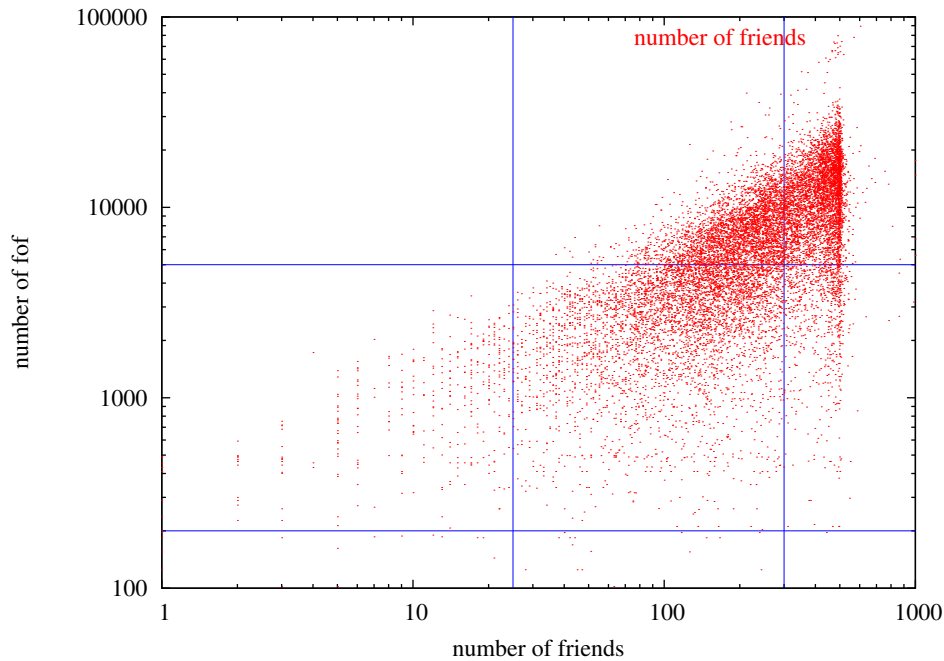


Figure 5.4: A scatter plot of the number of fofs versus the number of friends. The percentages of users in each of the nine areas divided by the lines are shown in Table 5.2.

have are found through the direct friend of the bootstrap user. This gives them all around 500 fofs. When a complete crawl of the network could have been made, this effect would disappear.

	<25 friends	25-300 friends	> 300 friends
> 5000 fofs	0.00%	25.98%	32.43%
200-5000 fofs	2.61%	33.41%	5.34%
< 200 fofs	0.107%	0.114%	0.007%

Table 5.2: The percentages of users that have certain numbers of friends and fofs. The ranges are shown by lines in Figure 5.4.

In Figure 5.4 the number of fofs are scattered against the number of direct friends. From this plot it is clear that a user needs a high number of friends to have many fofs. But still there are few users with many friends but few fofs. These are typically the users near the sides of the graph of which the friends have not yet been crawled.

Figure 5.4 has been divided into 9 regions. The percentages of users in each region are shown in Table 5.2.

The set of fofs of a user and the set of his direct friends are often not disjoint. In that case, some friends of this user are friends of each other and thus both friends

and fofs. This form of clustering is analyzed in Section 5.3.3

5.3.3 Clustering coefficient

To find to what extent the Friendster network is clustered and not just a randomly connected network, we will analyze the clustering coefficient. Loosely speaking, the clustering coefficient measures how close each user's neighborhood is to a clique.

To define the clustering coefficient, it is necessary to describe the crawled network as a directed graph $G = (V, E)$. The set of vertices $V = \{v_1, \dots, v_n\}$ consists of all Friendster users in our network. The set of directed edges E , consists of the crawled directed friendship relations. A directed edge from v_i to v_j will be denoted as e_{ij} . In our graph, the size of V is equal to 14,010 and the size of E is equal to 3,608,981 directed edges. The clustering coefficient of some vertex v_i in the graph is defined by:

$$C_i = \frac{|\{e_{jk}\}|}{k_i(k_i - 1)} : v_j, v_k \in N_i, e_{jk} \in E, \quad (5.1)$$

where N_i is the neighborhood of vertex v_i , defined by

$$N_i = \{v_j : e_{ij} \in E\}, \quad (5.2)$$

and k_i is the size of N_i . The clustering coefficient C of the whole network is defined as the average clustering coefficient of all its vertices, that is:

$$C = \frac{1}{n} \sum_{i=1}^n C_i. \quad (5.3)$$

There are different definitions of how to treat the vertices with a neighborhood of size one and zero. They have an undefined clustering coefficient. In our calculations, we gave these vertices a clustering coefficient of zero, and included them in the average of Equation 5.3. Others omit these vertices or give them a C_i equal to 1; the clustering coefficient of the whole network is then higher.

We will take the bootstrap user as an example. There exist 3,112 friendships between the 600 direct friends of the bootstrap user. His clustering coefficient is 0.008573. This means that the bootstrap user has 0.86 percent of the friendships that all his friends could have when they would form a completely connected graph. The clustering coefficient C of the complete network is equal to 0.0320. To compare this to a randomly generated network, we have created such a random network with the same number of vertices and edges as in the Friendster graph. The created network is not completely random. It was created by first connecting all vertices in a circle and then adding random edges until the wanted number of edges are in the graph. This sort of network guarantees that all vertices are connected with at least two others, so that the clustering coefficient is defined for each vertex. In Figure 5.5 an example of such a network is depicted.

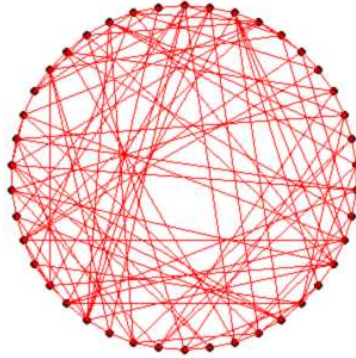


Figure 5.5: A randomly generated network except for the connections of all vertices to their neighbors in the circle. The network has 42 vertices and 252 edges.

The clustering coefficient of a random network can also be easily calculated. The clustering coefficient is defined as the number of edges between the neighbors of a vertex divided by the number of edges there could have been if the neighbors were completely connected. Therefore, it is equal to the probability p that each of the possible edges in the whole network has been placed. A completely connected directed graph of v vertices has $v(v - 1)$ edges. So, the clustering coefficient C_r of a random network with v vertices and e edges is equal to:

$$C_r = p = \frac{e}{v(v - 1)}. \quad (5.4)$$

For a network with same properties as the Friendster network, C_r is equal to 0.0184. Our simulation gives exactly the same clustering coefficient. The difference between C and C_r shows that the Friendster network is clearly more clustered. Since the Friendster crawl was only partial, the difference is not that big. Like the fof numbers, also the clustering coefficient is affected by the fact that a big part of the users were not yet crawled. There exist many users with most of their friends still in the queue, so that these friends cannot have known friendships with each other. These users all get a clustering coefficient lower than they have in reality.

To show the difference between the distribution of a random network and the Friendster data, Figure 5.6 shows both probability density functions. The clustering coefficient of the Friendster data has much higher variance, than the random network. This shows that some parts of the Friendster network are highly clustered, leaving other parts with less edges than in the random network. Many of the vertices with low clustering coefficients can be explained by the reason mentioned above. In the random situation the neighborhood of every vertex is more or less the same, leading to a narrow probability density function.

Club Nexus is a social network that enabled students and employees of Stanford University to keep in touch. It was designed by Orkut Buyukkokten, who later initiated the Orkut social network. Buyukkokten and others analyzed the Club Nexus

network in 2002 [13]. It consists of 2,469 Nexus users and 10,119 links between them. This gives an average of only 4 friends per user (8 directed friendships). With these small number of friends, it is easier to obtain a higher clustering coefficient caused by many small clusters. The clustering coefficient of the Nexus community was 0.17. According to the authors, this is 40 times higher than a random network with the same number of vertices and edges. With our simulation, a random network with these properties will give a clustering coefficient equal to 0.00369, which is even 46 times smaller. That big a difference cannot be shown in our data.

In Figure 5.7 it shows that users with high clustering coefficients most often have only few friends. This means that there exist small, highly connected clusters, but bigger clusters are more sparsely connected. This can show that many Friendster users have so many virtual friends, with so many fofs, that one need to invest enormous amounts of energy to get really acquainted with those people and to form a cluster by creating lots of friendships.

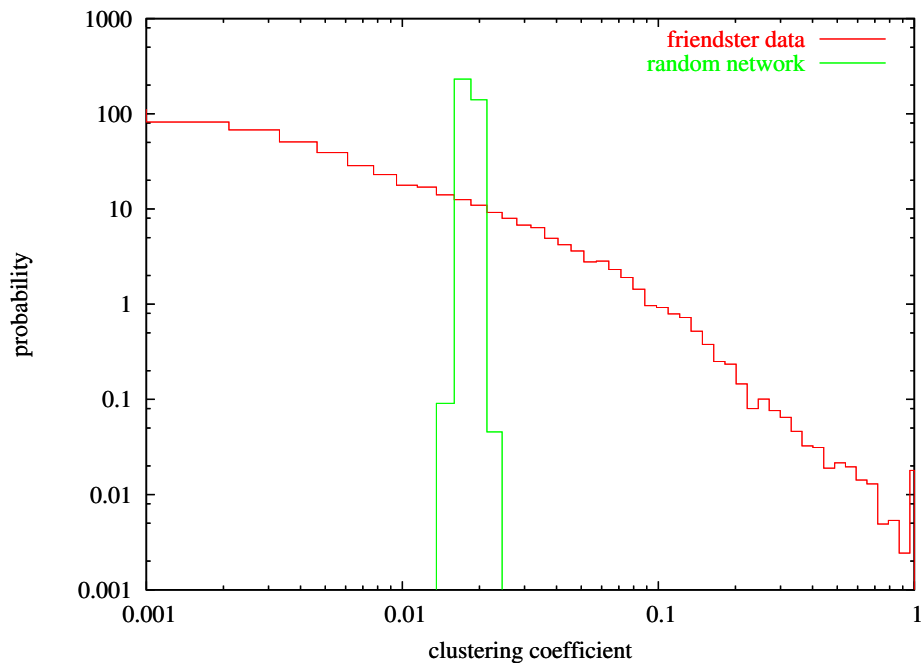


Figure 5.6: The probability density function of the clustering coefficient of the vertices in the Friendster network and a random network with an equal number of vertices and edges.

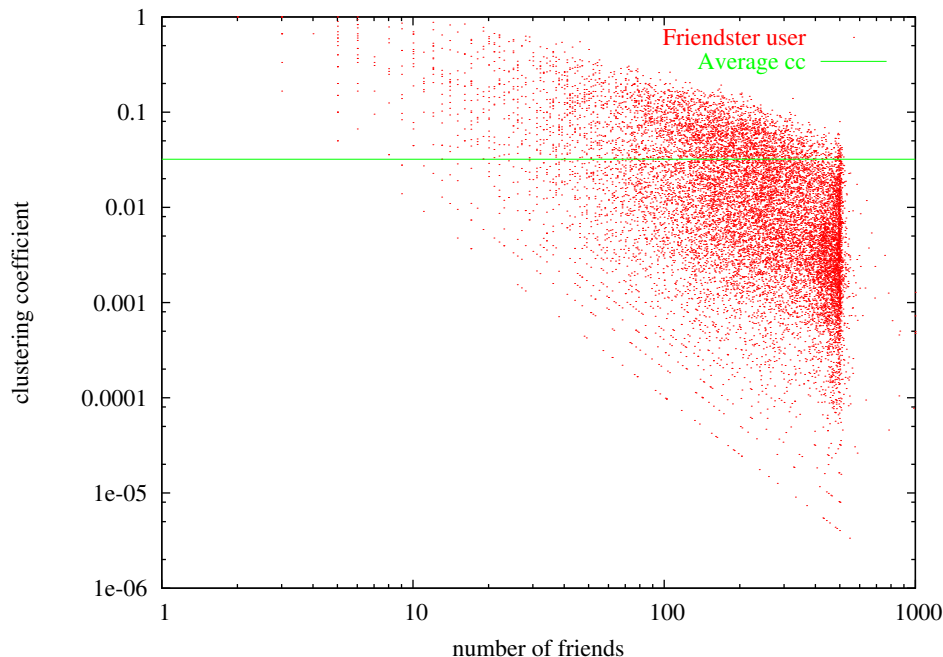


Figure 5.7: A scatter plot of the clustering coefficient versus the number of friends. The horizontal line denotes the average clustering coefficient of the network.

5.4 User exchange with Bloom filters

Now that we have analyzed the Friendster social network, we can give realistic properties of Bloom filters when they are used to exchange sets containing friends of fofs. The exchange of these sets is done to retrieve information about which friends/fofs are currently online, to stimulate clustering by exchanging interesting friends, or to find which friends you share with other peers.

5.4.1 Mutual friend discovery

In Section 4.1.2 the process of finding friends that you share with other friends is explained. The Friendster data shows that clustering really exists between friends so that a peer can take advantage of this. With the network properties collected from the Friendster crawl, we will estimate the sizes of Bloom filters used for mutual friend discovery and thereby the bandwidth usage of the process.

Assume peer p_1 has 260 friends and a clustering coefficient equal to 0.02 (both average numbers found in the crawled data). Then, according to Equation 5.1, there exist 1,347 directed friendships between the friends of p_1 . This equals 673 normal friendships and approximately 5 'inter-friend' friendships per friend.

When we use Bloom filters to broadcast the friends of p_1 , we need to choose a reasonable false-positive probability. In the process of mutual friend discovery,

errors do not have severe effects. When p_2 , a receiver of the friends of p_1 , falsely concludes that some peer is a friend of p_1 , it might add this peer to his response Bloom filter. But p_1 will never notice this, because it only tests for the membership of his friends on the received Bloom filter. A small bandwidth loss is the only effect for p_1 . For the receiving peer p_2 , this false positive will result in erroneous mutual-friends information.

The occurrence of false positives in the response has however a negative effect for p_1 . Such a false positive will make p_1 falsely conclude that some mutual friend exists. Because p_1 is the initiator of the mutual-friends request, the mutual-friends information should be more accurate. So false positives in response Bloom filters should be less probable than in the request Bloom filters. We will assume that a probability of 0.0216 is accurate enough for the request Bloom filter, the responses will use $f = 4.59 \cdot 10^{-4}$. In Table 2.1 we show that we need 8 and 16 bits per element, respectively, to achieve these false-positive probabilities.

The bit array of the Bloom filter with the 260 friends of p_1 will thus be only 260 bytes long. Including 100 bytes header information that defines the used hash-functions, salts and format of the Bloom filter, the total can be send in 360 bytes. If the receiver tests all his 260 friends on the Bloom filter, the false-positive probability of 0.0216 will give around 5 false positives. The response will contain these 5 users plus the 5 genuine mutual friends. The bit array of the Bloom filter storing these users will be only be 20 bytes long. Leading to a packet of 120 bytes.

When half of the users initiate this communication daily to update their mutual-friends list, it will use up 62.4 KByte per day per user. This kind of communication is completely negligible compared to the multiple Gigabyte shared content on file-sharing networks like Bittorrent.

In Table 5.3 this calculation is done for users with different number of friends. The clustering coefficients are read from Figure 5.7. F_{req} is the Bloom filter with the mutual friends request, F_{resp} the filter with the response. In the calculation of the total bandwidth usage, the header size of every sent Bloom filter is assumed to be 100 bytes and it is assumed that only half your friends has to be queried, because one request will give mutual-friend information to two peers. The error probabilities of the Bloom filters are equal to the filters mentioned above.

Number of friends	10	100	260	600
Clustering coefficient	0.3	0.03	0.02	0.0025
Size F_{req} in bytes	10	100	260	600
Size F_{resp} in bytes	15	16	20	13
Bandwidth usage in Kbytes	1.125	15.8	62.4	243.9

Table 5.3: The Bloom filter sizes and bandwidth usage of different users that update the mutual-friends information for all their friends.

5.4.2 Online peer discovery

Based on the Friendster data and calculation in Section 5.4.1, we assume that an average peer p_1 has 260 friends and there exist 1,078 friendships between those friends.

Now let 10% of the friends of a peer p_1 be online and still connectable when it turns on its computer. Then p_1 can use these 26 friends to discover the status of the others. Let 25% of the 260 friends be online, but unconnectible because they changed their ip address or port number. Because p_1 has recently stored mutual-friend information of its friends, it knows which of the 234 unconnectible friends it shares with the 26 online friends. A Bloom filter with the identifiers of these (approximately 108) users will be sent as an online peer request. The online friends will respond with raw lists of the connection information and peer identifiers of the requested friends that are online.

The number of friends that can be found through this method depends on when all peers came online and if/when they changed their connection addresses. The sizes of Bloom filters also depend on these properties. A fact is, the more every peer knows about its social neighborhood, the better it is able to ask the right friends for the right information.

Instead of only asking for the whereabouts of its social neighborhood, a peer can also requests connection information of all peers that it knows of, but can not reach on this moment. The amount of peers in the peer cache can be thousands, so it seems prudent to choose a small Bloom filter with a relatively big error. The response to an online peer request will be the same as above: raw lists of peer identifiers and connection information. False positives lead in this case to the reception of peer identifiers and connection information of peers that are unknown to the requester. These can be easily filtered out or newly be added to the cache.

To show that mutual-friend information is needed to efficiently request who is online, here is an example that does not use this information. Imagine a network in which peers intensively check which other peers are online. Therefore, every peer checks every hour what peers in his cache are connectible by sending a Bloom filter of the identifiers to all his friends and fofs. We assume that the most recent 10,000 peer identifiers will be put in a Bloom filter of 8 bits per element and send to 750 online fofs around you. Only the requests consume 7.5 MByte bandwidth per user per hour, which is too much overhead. Sending only the identifiers of peers that each of your fofs could know the connection information of, reduces this overhead. The size of the responses will be the same in both cases. Raw lists of identifiers, ip addresses and ports, which sizes will depend on how many people are online.

To get more insight in the bandwidth cost of online peer discovery, more experiments have to be done. More information is needed about the uptime of peers and how often they change their connection information. Also the size and properties of the peer cache will influence the process of finding online peers. One can think of the effect that a peers that are often online will automatically be trusted more and thus more often queried for online information. This would create some sort

of automatic super-peer selection that could be helpful in finding online peers.

Chapter 6

Conclusions and Future Work

In this report we have given a literature overview over Bloom filters and their applications. We have shown that a Bloom filter is an efficient data structure that can record set membership, while reducing both memory usage and computation in several applications. Bloom filters offer the flexibility to make a trade-off between their size and the probability for false positives when checking for the membership of elements. The efficiency that is gained by the smaller size of a Bloom filter often outnumbers the efficiency loss introduced by membership errors. Bloom filters can be used as a basis for more complex data structures, such as the Bloomier filter or the Counting Bloom filter. When the transmission size of the Bloom filter is most important, one can choose to compress it at the cost of computation complexity and memory usage.

Bloom filters have found broad use in various applications. Most systems, like Web caching and spell checkers, use Bloom filters because they can store or transmit large sets efficiently. Another valuable property of a Bloom filter is its ability to keep the stored data private from anybody not owning the elements in the filter. The privacy protection is caused by the secure hashing of elements, leading to hash values that can only be interpreted by users that have the elements.

When important data sets are transmitted in a p2p file-sharing system through Bloom filters, it is interesting to know how secure they are. We have discussed methods to make it impossible to transmit a fake Bloom filter to pretend that you own certain content or peer identifiers.

We have also analyzed the properties of the Friendster social network. Using a Web crawler, a big portion of the Friendster network has been crawled and saved as a directed graph. The properties of this Friendster graph, including the number of friends, number of unique friends of a friend and clustering coefficient of each user, is used to get a notion of how a social p2p file-sharing network would function. Furthermore, we have discussed some applications of Bloom filters in such a file-sharing network. The Friendster network shows that users actively create friendships with many people (258 friends per user on average) and therefore know many other users through just a few steps in the network. This fact is promising

for the hypothesis that a social network can be used to create a stable and efficient p2p file-sharing network.

Open research questions that we intend to address during the coming months include:

- How effective is online peer discovery in a social file-sharing network? Is it possible to create a stable online peer discovery mechanism using the social neighborhood of a peer? How much bandwidth and computational overhead does this mechanism cost? How does its stability react to the number of online sessions and the session length of users? How can peers that are trusted by many friends (for instance, because they are often online) be protected against a flood of queries from their social neighborhood?
- In what ways can a peer attack his friends if a social network is used as a basis for a p2p file-sharing network? What can be done to protect a file-sharing network from those attacks? Can Bloom filters be used to protect (the information on) peers?
- What are other applications for Bloom filters in p2p file-sharing networks? Can existing features in file-sharing networks be made more efficient or private with Bloom filters?

Bibliography

- [1] aSmallWorld. <http://www.asmallworld.net>.
- [2] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Comm. of the ACM*, 13(7):422–426, 1970.
- [3] title = Bittorrent protocol, <http://www.bittorrent.com/protocol/html> Bram Cohen.
- [4] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. In *Proc. of the 40th Annual Allerton Conference on Communication, Control and Computing*, pages 636 – 646, Illinois, USA, 2002.
- [5] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. The bloomier filter: an efficient data structure for static support lookup tables. In *SODA '04: Proc. of the Fifteenth Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 30–39, Philadelphia, PA, USA, 2004.
- [6] Saar Cohen and Yossi Matias. Spectral bloom filters. In *SIGMOD '03: Proc. of the 2003 ACM SIGMOD Int. conference on Management of data*, pages 241–252, New York, NY, USA, 2003. ACM Press.
- [7] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep packet inspection using parallel bloom filters. *Micro, IEEE*, 24(1):52 – 61, 2003.
- [8] Henrich Eisenhardt, Muller. Spektrale bloom-filter fur peer-to-peer information retrieval. In *Proceedings Informatik 2004*, pages 44–48, 2004.
- [9] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, 2000.
- [10] Friendster. <http://www.friendster.com>.
- [11] Maciej Cegłowski Joshua Schachter. Loaf, a simple email extension for contact recognition using bloomfilter, <http://loaf.cantbedone.org>, 2004.
- [12] John Kubiawicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Chris Wells, and Ben Zhao. Oceanstore: an architecture for global-scale persistent storage. In *ASPLOS-IX: Proc. of the ninth international conference on Architectural support for programming languages and operating systems*, pages 190–201, New York, NY, USA, 2000. ACM Press.
- [13] O. Buyukkokten L.A. Adamic and E. Adar. A social network caught in the web. *First Monday*, 8(6), 2003.
- [14] Cache Logic. <http://www.cachelogic.com>.
- [15] Udi Manber and Sun Wu. An algorithm for approximate membership checking with application to password security. *Inf. Process. Lett.*, 50(4):191–197, 1994.
- [16] M. McIlroy. Development of a spelling list. *Communications, IEEE Transactions on*, 30(1):91–99, 1982.
- [17] ICQ Instant Messenger. <http://www.icq.com>, 2005.
- [18] S. Milgram. The small world problem. *Psychology Today*, 1967.

- [19] Kirsch Mitzenmacher. Building a better bloom filter. Unpublished., feb 2005.
- [20] Kirsch Mitzenmacher. Distance-sensitive bloom filters. Unpublished., feb 2005.
- [21] Michael Mitzenmacher. Compressed bloom filters. *IEEE/ACM Trans. Netw.*, 10(5):604–612, 2002.
- [22] Janus Friis Niklas Zennström. Skype, <http://www.skype.com>.
- [23] Orkut. <http://www.orkut.com>.
- [24] P. Reynolds and A. Vahdat. Efficient peer-to-peer keyword searching.
- [25] Sean C. Rhea and John Kubiawicz. Probabilistic location and routing. In *Proc. of INFOCOM 2002. IEEE*, volume 3, pages 1248–1257, 2002.
- [26] ICP working group. (1998). National lab for applied network research.
- [27] J. Wang Yifeng Zhu, Hong Jiang. Hierarchical bloom filter arrays (hba): a novel, scalable metadata management system for large cluster-based storage. In *IEEE International Conference on Cluster Computing*, pages 165 – 174, 2004.