Internet Mathematics Vol. 1, No. 4: 485-509

Network Applications of Bloom Filters: A Survey

Andrei Broder and Michael Mitzenmacher

Abstract. A Bloom filter is a simple space-efficient randomized data structure for representing a set in order to support membership queries. Bloom filters allow false positives but the space savings often outweigh this drawback when the probability of an error is controlled. Bloom filters have been used in database applications since the 1970s, but only in recent years have they become popular in the networking literature. The aim of this paper is to survey the ways in which Bloom filters have been used and modified in a variety of network problems, with the aim of providing a unified mathematical and practical framework for understanding them and stimulating their use in future applications.

1. Introduction

A Bloom filter is a simple space-efficient randomized data structure for representing a set in order to support membership queries. Bloom filters allow false positives but the space savings often outweigh this drawback when the probability of an error is made sufficiently low. Burton Bloom introduced Bloom filters in the 1970s [Bloom 70], and ever since they have been very popular in database applications. Recently they started receiving more widespread attention in the networking literature.

In this paper, we survey the ways in which Bloom filters have been used and modified for a variety of network problems, with the aim of providing a unified mathematical and practical framework for them and stimulating their use in future applications. We first describe the mathematics behind Bloom filters,

© A K Peters, Ltd.
1542-7951/04 $0.50 per page

their history, and some important variations. We then consider four types of network-related applications of Bloom filters:

- Collaborating in overlay and peer-to-peer networks: Bloom filters can be used for summarizing content to aid collaborations in overlay and peer-to-peer networks.
- Resource routing: Bloom filters allow probabilistic algorithms for locating resources.
- Packet routing: Bloom filters provide a means to speed up or simplify packet routing protocols.
- Measurement: Bloom filters provide a useful tool for measurement infrastructures used to create data summaries in routers or other network devices.

We emphasize that this simple categorization is very loose; some applications fit into more than one of these categories, and these categories are not meant to be exhaustive. Indeed, we suspect that new applications of Bloom filters and their variants will continue to "bloom" in the network literature. Also, we emphasize that we are providing only brief summaries of the work of many others.

The theme unifying these diverse applications is that a Bloom filter offers a succinct way to represent a set or a list of items. There are many places in a network where one might like to keep or send a list, but a complete list requires too much space. A Bloom filter offers a representation that can dramatically reduce space, at the cost of introducing false positives. If false positives do not cause significant problems, the Bloom filter may provide improved performance. We call this the Bloom filter principle, and we repeat it for emphasis below.

The Bloom filter principle: *Wherever a list or set is used, and space is at a premium, consider using a Bloom filter if the effect of false positives can be mitigated.*

2. Bloom Filters: Mathematical Preliminaries

2.1. Standard Bloom Filters

We begin by presenting the mathematics behind Bloom filters. A Bloom filter for representing a set $S = \{x_1, x_2, \ldots, x_n\}$ of n elements is described by an array of m bits, initially all set to 0. A Bloom filter uses k independent hash functions $h_1, \ldots, h_k$ with range $\{1, \ldots, m\}$. For mathematical convenience, we make the

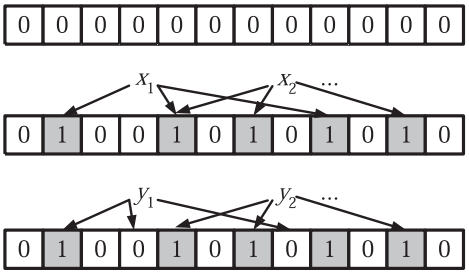

Figure 1. An example of a Bloom filter. The filter begins as an array of all 0s. Each item in the set x_i is hashed k times, with each hash yielding a bit location; these bits are set to 1. To check if an element y is in the set, hash it k times and check the corresponding bits. The element y_1 cannot be in the set, since a 0 is found at one of the bits. The element y_2 is either in the set or the filter has yielded a false positive.

natural assumption that these hash functions map each item in the universe to a random number uniform over the range $\{1, \ldots, m\}$. For each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$. A location can be set to 1 multiple times, but only the first change has an effect. To check if an item y is in S, we check whether all $h_i(y)$ are set to 1. If not, then clearly y is not a member of S. If all $h_i(y)$ are set to 1, we assume that y is in S, although we are wrong with some probability. Hence, a Bloom filter may yield a *false positive*, where it suggests that an element x is in S even though it is not. Figure 1 provides an example. For many applications, false positives may be acceptable as long as their probability is sufficiently small. To avoid trivialities we will silently assume from now on that $kn < m$.

The probability of a false positive for an element not in the set, or the *false positive rate*, can be estimated in a straightforward fashion, given our assumption that hash functions are perfectly random.[1] After all the elements of S are hashed into the Bloom filter, the probability that a specific bit is still 0 is

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

We let $p = \mathrm{e}^{-kn/m}$, and note that p is a convenient and very close (within $O(1/m)$) approximation for p'.

Now, let ρ be the proportion of 0 bits after all the n elements are inserted in the table. The expected value for ρ is of course $\mathbb{E}(\rho) = p'$. Conditioned on ρ,

[1] Early work considering the performance of Bloom filters with practical hash functions was done by Ramakrishna [Ramakrishna 89]. The question of what hash function to use in practice remains an interesting open question; currently MD5 is a popular choice [Fan et al. 00].

the probability of a false positive is

$$(1-\rho)^k \approx (1-p')^k \approx (1-p)^k.$$

We already discussed the second approximation; the first one is justified by the fact that with high probability ρ is very close to its mean. We will return to this fact at the end of this section.

We let

$$f' = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k = (1-p')^k$$

and

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

In general, it is often easier to use the asymptotic approximations p and f in analysis, rather than p' and f'.

It is worth noting that sometimes Bloom filters are described slightly differently: instead of having one array of size m shared among all the hash functions, each hash function has a range of m/k consecutive bit locations disjoint from all the others. The total number of bits is still m, but the bits are divided equally among the k hash functions. Repeating the analysis above, we find that in this case the probability that a specific bit is 0 is

$$\left(1 - \frac{k}{m}\right)^n \approx \mathrm{e}^{-kn/m}.$$

Asymptotically, then, the performance is the same as the original scheme. However, since for $k \geq 1$

$$\left(1 - \frac{k}{m}\right)^n \leq \left(1 - \frac{1}{m}\right)^{kn}$$

(with equality only when $k = 1$), the probability of a false positive is actually always at least as large with this division. Since the difference is small, this approach may still be useful for implementation reasons; for example, dividing the bits among the hash functions may make parallelization of array accesses easier.

Suppose that we are given m and n and we wish to optimize for the number of hash functions. There are two competing forces: using more hash functions gives us more chances to find a 0 bit for an element that is not a member of S, but using fewer hash functions increases the fraction of 0 bits in the array. The optimal number of hash functions that minimizes f as a function of k is easily found by taking the derivative. More conveniently, note that $f = \exp(k \ln(1 - \mathrm{e}^{-kn/m}))$.

Let $g = k\ln(1 - \mathrm{e}^{-kn/m})$. Minimizing the false positive rate f is equivalent to minimizing g with respect to k. We find that

$$\frac{\partial g}{\partial k} = \ln\left(1 - \mathrm{e}^{-\frac{kn}{m}}\right) + \frac{kn}{m}\frac{\mathrm{e}^{-\frac{kn}{m}}}{1 - \mathrm{e}^{-\frac{kn}{m}}}.$$

It is easy to check that the derivative is 0 when $k = \ln 2 \cdot (m/n)$; further efforts reveal that this is a global minimum. Alternatively, using $p = \mathrm{e}^{-kn/m}$, we find that

$$g = -\frac{m}{n}\ln(p)\ln(1-p),$$

from which symmetry reveals that the minimum value for g occurs when $p = 1/2$, or equivalently $k = \ln 2 \cdot (m/n)$. In this case the false positive rate f is $(1/2)^k \approx (0.6185)^{m/n}$. In practice, of course, k must be an integer, and a smaller, sub-optimal k might be preferred since this reduces the number of hash functions that have to be computed.

The pleasant result that p equals $1/2$ when the probability of a false positive is minimized does not depend on the asymptotic approximation. Repeating the argument above, for $f' = \exp(k\ln(1-(1-1/m)^{kn}))$, $g' = k\ln(1-(1-1/m)^{kn})$, and $p' = (1-1/m)^{kn}$, we have

$$g' = \frac{1}{n\ln(1-1/m)}\ln(p')\ln(1-p'),$$

and again by symmetry g' and hence f' are minimized when $p' = 1/2$. (One could similarly derive this fact via calculus.)

Finally, the astute reader may be concerned that the fraction of 0 bits in the Bloom filter array may not equal p' (or p) on any given instantiation; the number of 0s and 1s in the Bloom filter depends on the results of the hashing. Indeed, p' simply represents the *expected* fraction of 0 bits in the array. If the number of 0 bits in the array is substantially less than expected, then the probability of a false positive will be higher than the quantity f that we computed. Mitzenmacher shows that, in fact, the fraction of 0 bits is extremely concentrated around its expectation, using a simple martingale argument [Mitzenmacher 02]. Specifically, if X is a random variable corresponding to the number of 0 bits in a Bloom filter, then one can show, using the Azuma-Hoeffding inequality, that for any $\epsilon > 0$,

$$\mathbb{P}(|X - p'm| \geq \epsilon m) \leq 2\mathrm{e}^{-2\epsilon^2 m^2/nk}.$$

Similar bounds can be had by making use of negative dependence [Dubhasi and Ranjan 98], which corresponds to the intuitive idea that when one bit is set to 1, it (slightly) lowers the probability that every other bit is set to 1. Negative

dependence allows Chernoff bounds to be applied to bound the fraction of 0 bits, giving a similar exponential tail bound. Hence, while on any specific Bloom filter the fraction of 0 bits may not be exactly p', with high probability it will be very close to p' for large arrays, which justifies the use of p' (and p) in our analyses above.

2.2. A Lower Bound

One means of determining how efficient Bloom filters are is to consider how many bits m are necessary to represent all sets of n elements from a universe in a manner that allows false positives for at most a fraction ϵ of the universe but allows no false negatives. We derive a simple lower bound on m for this case.

Suppose that our universe has size u. Our representation must associate an m-bit string with each of the $\binom{u}{n}$ possible sets. For each set X, let $F(X)$ be the string to which the set is mapped in our representation. We say that an m-bit string s *accepts* an element x of the universe if $s = F(X)$ for some X containing x; that is, there is some set in the universe containing x for which s is the representative string. Otherwise, s *rejects* x. Intuitively, if s accepts x, then given s we should conclude that the set that generated s contains x, and if s rejects x, we can be sure that the set that generated x does not contain s.

Consider a specific set X of n elements. Any string s that is used to represent X must accept every element x of X (remember, no false negatives!), but it may also accept $\epsilon(u-n)$ other elements of the universe while maintaining a false positive rate of at most ϵ. Each string s therefore accepts at most $n+\epsilon(u-n)$ elements. A fixed string s can be used to represent any of the $\binom{n+\epsilon(u-n)}{n}$ subsets of size n of these elements, but it cannot be used to represent any other sets. If we use m bits, then we have 2^m distinct strings that must represent all the $\binom{u}{n}$ sets. Hence, we must have

$$2^m \binom{n+\epsilon(u-n)}{n} \geq \binom{u}{n},$$

or

$$m \geq \log_2 \frac{\binom{u}{n}}{\binom{n+\epsilon(u-n)}{n}} \approx \log_2 \frac{\binom{u}{n}}{\binom{\epsilon u}{n}} \geq \log_2 \epsilon^{-n} = n \log_2(1/\epsilon).$$

The approximation above is suitable when n is small compared to ϵu, which is the practical case of interest. We, therefore, find that m needs to be essentially $n \log_2(1/\epsilon)$ for any representation scheme with a false positive rate bounded by ϵ.

Recall that the (expected) false positive rate f for a Bloom filter is

$$f = (1/2)^k \geq (1/2)^{m \ln 2/n},$$

since the optimal value for k is $m \ln 2/n$. After some algebraic manipulation, we find that $f \leq \epsilon$ requires

$$m \geq n \frac{\log_2(1/\epsilon)}{\ln 2} = n \log_2 \mathrm{e} \cdot \log_2(1/\epsilon).$$

Thus, space-wise Bloom filters are within a factor of $\log_2 \mathrm{e} \approx 1.44$ of the (asymptotic) lower bound. Alternatively, keeping the space constant, if we use $n \cdot j$ bits for the table, optimal Bloom filters will yield a false positive rate of about $(0.6185)^j$, while the lower bound error rate is only $(0.5)^j$.

There are methods to represent sets that use fewer bits than Bloom filters while maintaining the same rate of false positives, including the compressed Bloom filters discussed in Section 2.6 and techniques based on perfect hashing. Such schemes, however, appear more complicated, require more computation, and offer less flexibility than Bloom filters.

For example, if the set X of n elements is fixed, one can find a perfect hash function for X, say $h_X : [1..u] \to [1..n]$, plus a fully uniform random hash function $\phi : [1..u] \to [0..2^j - 1]$. Then for each $x \in X$, we store $\phi(x)$ at location $h_X(x)$ in a table with n entries of j bits each. Clearly, this scheme matches the lower bound since $m = n \cdot j$, and the probability of a false positive is exactly $1/2^j$. On the other hand, any change in the set X would require an expensive recomputation of a perfect hash function.

2.3. Hashing vs. Bloom Filters

Hashing is among the most common ways to represent sets. Each item of the set is hashed into $\Theta(\log n)$ bits, and a (sorted) list of hash values then represents the set. This approach yields very small error probabilities. For example, using $2 \log_2 n$ bits per set element, the probability that two distinct elements yield the same hash value is $1/n^2$. Hence, the probability that any element not in the set matches some hash value in the set is at most $n/n^2 = 1/n$ by the standard union bound.

Bloom filters can be interpreted as a natural generalization of hashing that allows more interesting tradeoffs between the number of bits used per set element and the probability of false positives. (Indeed, a Bloom filter with just one hash function is equivalent to ordinary hashing.) Bloom filters yield a constant false positive probability even for a constant number of bits per set element. For example, when $m = 8n$, the false positive probability is just over 0.02. For most theoretical analyses, this tradeoff is not useful: typically, one needs an asymptotically vanishing probability of error, which is achievable only when we use $\Theta(\log n)$ bits per element. Hence, Bloom filters have received little attention in the theory community. In contrast, for practical applications, a constant false

positive probability may well be worthwhile in order to keep the number of bits per element constant.

2.4. Standard Bloom Filter Tricks

The simple structure of Bloom filters makes certain operations very easy to implement. For example, suppose that one has two Bloom filters representing sets S_1 and S_2 with the same number of bits and using the same hash functions. Then, a Bloom filter that represents the union of two sets can be obtained by taking the OR of the two bit vectors of the original Bloom filters.

Another nice feature is that Bloom filters can easily be halved in size, allowing an application to dynamically shrink a Bloom filter. Suppose that the size of the filter is a power of 2. To halve the size of the filter, just OR the first and second halves together. When hashing to do a lookup, the highest order bit can be masked.

Bloom filters can also be used to approximate the intersection between two sets. Again, suppose that one has two Bloom filters representing sets S_1 and S_2 with the same number of bits and using the same hash functions. Intuitively, the inner product of the two bit vectors is a measure of their similarity. More formally, the jth bit will be set in both filters if it is set by some element in $S_1 \cap S_2$, or if it is set simultaneously by some element in $S_1 - (S_1 \cap S_2)$ and by another element in $S_2 - (S_1 \cap S_2)$. The probability that the jth bit is set in both filters is therefore

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

After some algebraic simplification, we find that the expected magnitude of the inner product of the two Bloom filters is therefore

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

Hence, given $|S_1|$, $|S_2|$, k, m, and the magnitude of the inner product, one can calculate an estimate of the intersection $|S_1 \cap S_2|$ using the equation above. Note that if $|S_1|$ and $|S_2|$ are not given, they can also be estimated by counting the number of 0 bits in the Bloom filters for S_1 and S_2, since as explained in Section 2.1, the number of 0 bits for a set S is strongly concentrated around its expectation $m(1 - 1/m)^{k|S|}$.

Letting Z_1 (respectively Z_2) be the number of 0s in the filter for S_1 (respectively S_2) and Z_{12} be the number of 0s in the inner product, we obtain that

$$\frac{1}{m}\left(1-\frac{1}{m}\right)^{-k|S_1\cap S_2|} \approx \frac{Z_1+Z_2-Z_{12}}{Z_1 Z_2}.$$

2.5. Counting Bloom Filters

Suppose that we have a set that is changing over time, with elements being inserted and deleted. Inserting elements into a Bloom filter is easy; hash the element k times and set the bits to 1. Unfortunately, one cannot perform a deletion by reversing the process. If we hash the element to be deleted and set the corresponding bits to 0, we may be setting a location to 0 that is hashed to by some other element in the set. In this case, the Bloom filter no longer correctly reflects all elements in the set.

To avoid this problem, Fan et al. [Fan et al. 00] introduced the idea of a *counting Bloom filter*.[2] In a counting Bloom filter, each entry in the Bloom filter is not a single bit but rather a small counter. When an item is inserted, the corresponding counters are incremented; when an item is deleted, the corresponding counters are decremented. To avoid counter overflow, we choose sufficiently large counters.

The analysis from [Fan et al. 00] reveals that 4 bits per counter should suffice for most applications. To determine a good counter size, consider a counting Bloom filter for a set with n elements, k hash functions, and m counters. Let $c(i)$ be the count associated with the ith counter. The probability that the ith counter is incremented j times is a binomial random variable:

$$\mathbb{P}(c(i)=j)=\binom{nk}{j}\left(\frac{1}{m}\right)^{j}\left(1-\frac{1}{m}\right)^{nk-j}.$$

The probability that any counter is at least j is bounded above by $m\mathbb{P}(c(i)\geq j)$, which can be calculated using the above formula.

A loose but useful bound can also be derived as follows:

$$\mathbb{P}(c(i)\geq j)\leq\binom{nk}{j}\frac{1}{m^j}\leq\left(\frac{enk}{jm}\right)^{j}.$$

Suppose that we restrict ourselves to $k\leq(\ln 2)m/n$, since we have argued that we can optimize the false positive rate with $k=(\ln 2)m/n$. Then,

$$\mathbb{P}(\max_i c(i)\geq j)\leq m\left(\frac{e\ln 2}{j}\right)^{j}.$$

[2]The name "counting Bloom filter" for this data structure was introduced in [Mitzenmacher 02].

If we allow 4 bits per counter, the counter will overflow if and only if some counter reaches the value 16. From the above we have

$$\mathbb{P}(\max_i c(i) \geq 16) \leq 1.37 \times 10^{-15} \times m.$$

This bound holds for every set of at most n items, so a union bound says that a counting Bloom filter that represents t different sets of at most n items during its history overflows with probability at most $1.37 \times 10^{-15} \times mt$. This will suffice for most applications.

Another way of interpreting this result is to observe that when there are $m \ln 2$ total counter increments spread over m counters, then with high probability the maximum counter value is $O(\log m)$ and hence only $O(\log \log m)$ bits are necessary for each counter.

In practice, when a counter does overflow, one approach is to leave it at its maximum value. This can cause a later false negative only if eventually the counter goes down to 0 when it should have remained nonzero. If the deletions are random, the expected time to this event is relatively large.

2.6. Compressed Bloom Filters

In [Mitzenmacher 02], Mitzenmacher addresses the following question: suppose that a server is sending a Bloom filter to several other servers over a network. Can we gain anything by compressing the resulting Bloom filter? If we choose the optimal value for k to minimize the false probability as calculated previously, then $p = 1/2$. Under our assumption of good random hash functions, the bit array is essentially a random string of m 0s and 1s, with each entry being 0 or 1 with probability $1/2$. It would therefore seem that compression cannot offer any gain.

Mitzenmacher demonstrates the flaw in this reasoning. The problem is that we have optimized the false positive rate of the Bloom filter under the constraint that there are m bits in and n elements represented by the filter. Suppose instead that we optimize the false positive rate of the Bloom filter under the constraint that the number of bits to be sent *after compression* is z, but the size m of the array in its uncompressed form can be larger. It turns out that using a larger, but sparser, Bloom filter can yield the same false positive rate with a smaller number of transmitted bits. Alternatively, one can use the same number of transmitted bits but improve the false positive rate, or find a more suitable tradeoff between the two.

An example is given in Table 1, where the goal is to obtain small false positive rates while using less than 16 transmitted bits per element. Without compression, the optimal number of hash functions is 11, and the false positive probability

Array bits per element	m/n	16	28	48
Transmission bits per element	z/n	16	15.846	15.829
Hash functions	k	11	4	3
False positive probability	f	0.000459	0.000314	0.000222

Table 1. Using at most sixteen bits per element after compression, a bigger but sparser Bloom filter can reduce the false positive probability.

is 0.000459. By making a sparse Bloom filter using 48 bits per element but only 3 hash functions, one can compress the result down to less than 16 bits per item (with high probability) and decrease the false positive probability by roughly a factor of 2.

3. Historical Applications

3.1. Dictionaries

Bloom introduced Bloom filters in conjunction with an application to hyphenation programs [Bloom 70]. Most words can be hyphenated appropriately by applying a few simple rules. Some words, say around ten percent, require a table lookup. To avoid storing all the words that can be handled via the simple rules, Bloom suggests using a Bloom filter to keep a dictionary of words that require a lookup. False positives here cause words that could be handled via the simple rules to require a lookup.

Bloom filters were also used in early UNIX spell-checkers [McIlroy 82, Mullin and Margoliash 90]. Rather than store and search a dictionary, a Bloom filter representation of the dictionary was stored. A false positive could allow a misspelled word to be ignored. In early systems, where memory was a scarce and valuable resource, the space savings of a Bloom filter offered significant performance advantages.

Bloom filters were proposed as a means of succinctly storing a dictionary of unsuitable passwords for security purposes by Spafford [Spafford 92]. Manber and Wu describe a simple way to extend the technique so that passwords that are within edit distance 1 of the dictionary word are also not allowed [Manber and Wu 94]. In this setting, a false positive could force a user to avoid a password even if does not lie in the set of unsuitable passwords.

3.2. Databases

Bloom filters also found very early uses in databases. One use is to speed up semi-join operations [Bratbergsengen 84, Valdurez and Gardarin 84, Mackett and

Lohman 86, Mullin 90]. For example, suppose that one wanted to determine the employees of a business that live in cities where the cost of living is greater than 50,000 dollars. In a distributed database, one host might hold the information regarding the cost of living, while another might hold the information regarding where employees live. Rather than have the first database send a list of cities to the second, the first could send a Bloom filter of this list. The second host can then send a list of potential employee/city pairs back to the first database, where false positives can be removed. This has the potential to reduce overall communication between the two hosts. In a related vein, Bloom filters can be used to estimate the size of semi-join operations, using the fact that Bloom filters can be used to estimate intersections as described in Section 2.4 [Mullin 93].

Bloom filters can also be used for differential files [Gremilion 82, Mullin 83]. Suppose that all the changes to a database that occur during the day are processed as a batch job. A differential file then keeps track of changes that occur during the day. If one wants to read a record, one has to know if the record has been changed by some transaction in the differential file (in which case the differential file must be read) or if it can be read directly from the database. Instead of keeping a list of all records that have changed, one can keep a Bloom filter of records that have been changed. Here, a false positive forces one to read the differential file and the database even when a record has not been changed.

4. A Sample Network Application: Distributed Caching

To begin our survey of network applications, we present an early and especially instructive example of Bloom filters in a distributed protocol. Fan, Cao, Almeida, and Broder describe Summary Cache, which uses Bloom filters for Web cache sharing [Fan et al. 00]. In their setup, proxies cooperate in the following way: on a cache miss, a proxy attempts to determine if another proxy cache holds the desired Web page; if so, a request is made to that proxy rather than trying to obtain that page from the Web.

For such a scheme to be effective, proxies must know the contents of other proxy caches. In Summary Cache, to reduce message traffic, proxies do not transfer URL lists corresponding to the exact contents of their caches, but instead periodically broadcast Bloom filters that represent the contents of their cache. If a proxy wishes to determine if another proxy has a page in its cache, it checks the appropriate Bloom filter. In the case of a false positive, a proxy may request a page from another proxy, only to find that that proxy does not actually have that page cached. In that case, some additional delay has been incurred. In

this setting, false positives and false negatives may occur even without a Bloom filter, since the cache contents may change between periodic updates. The small additional chance of a false positive introduced by sending a Bloom filter is greatly outweighed by the significant reduction in network traffic achieved by using the succinct Bloom filter instead of sending the full list of cache contents. This technique is used in the open source Web proxy cache Squid, where the Bloom filters are referred to as Cache Digests [Rousskov and Wessels 98].

Since cache contents are changing frequently, [Fan et al. 00] suggests that caches use a counting Bloom filter to track their own cache contents, and broadcast the corresponding standard 0-1 Bloom filter to the other proxies. The alternative would be to construct a new Bloom filter from scratch whenever an update is sent; using the counting Bloom filter both reduces and amortizes this cost. Using delta compression and compressed Bloom filters, as described in [Mitzenmacher 02], can yield a further reduction in the number of bits transmitted.

5. Applications: P2P/Overlay Networks

An early peer-to-peer application of Bloom filters is due to Marais and Bharat [Marais and Bharat 97] in the context of a desktop web browsing assistant called *Vistabar*. Cooperative users of Vistabar store annotations and comments about the web pages that they visited in a central repository. Conversely, they see these comments whenever they load an annotated page. Rather than make a request for each URL encountered, Vistabar downloads periodically a Bloom filter corresponding to all annotated URLs.

5.1. Moderate-Sized P2P Networks

Many constructions for peer-to-peer networks use distributed hash tables in order to locate objects [Druschel and Rowstron 01, Ratnasamy et al. 01, Stoica et al. 01]. Distributed hash tables are particularly useful for large-scale scalability and for coping with peer-to-peer networks where individual nodes may frequently enter or leave the system.

For moderate-sized and more robust peer-to-peer systems of hundreds of nodes, Bloom filters may provide an attractive alternative for locating objects over distributed hash tables. While keeping a list of objects stored at every other node in a peer-to-peer system may be prohibitive, keeping a Bloom filter for every other node may be tractable. For example, instead of using a 64-bit identifier for each object, a Bloom filter could use 8 or 16 bits per object. False positives in this situation yield extraneous requests for objects to nodes that do not have them. A prototype P2P system dubbed PlanetP based on this idea is described in [Cuena-

Acuna et al. 02]; the filters actually store keywords associated with documents instead of object IDs. Implementation challenges include how frequently filters need to be updated.

In [Ledlie et al. 02], a similar approach that makes additional use of grouping and hierarchy is described. There, the idea is to introduce some hierarchy so that groups of nodes are governed by a leader. The leaders are meant to be more stable, long-lasting nodes that form a peer-to-peer network using Bloom filters in a manner similar to that described above, except that the Bloom filters cover objects held by the group. The group leader controls routing within a group and other group-specific issues.

5.2. Approximate Set Reconciliation for Content Delivery

Byers, Considine, Mitzenmacher, and Rost [Byers et al. 02a] demonstrate another area where Bloom filters can be useful in peer-to-peer applications. They suggest that peers may want to solve the following type of *approximate set reconciliation* problems. Suppose peer A has a set of items S_A, and peer B has a set of items S_B. Peer B would like to send peer A a succinct data structure so that A can start sending B items that B does not have, that is, items in $S_A - S_B$. One approach is to have B send A a Bloom filter; A then runs through its elements, checking each one against the Bloom filter, and sending any element that does not lie in S_B according to the filter. Because of false positives, not all elements in $S_A - S_B$ will be sent, but most will. The authors also consider an alternative data structure that uses Bloom filters, but allows for faster determination of elements in $S_A - S_B$ when the difference is small [Byers et al. 02b]. This work demonstrates that Bloom filters can also be useful as subroutines inside of more complex data structures.

The application [Byers et al. 02a] addresses the distribution of large files to many peers in overlay networks. The authors argue for encoded content. In this setting, peers may wish to collaborate during downloads, by receiving encoded packets from other peers as well as from the source, thus effectively increasing the download rate. If the encoded content is over a large alphabet, the problem of determining which encoded packets peer B needs that peer A has is simply the problem of determining $S_A - S_B$. Since the content is redundantly encoded, obtaining a large fraction of $S_A - S_B$ rather than the entire set is sufficient in this situation.

5.3. Set Intersection for Keyword Searches

Reynolds and Vahdat use Bloom filters in a similar fashion as [Byers et al. 02a], except that their goal is to find the set intersection instead of the set difference

[Reynolds and Vahdat 03]. Their approach is essentially the same as for database joins. Peer B can send a Bloom filter representing S_B to A; peer A then sends the elements of S_A that appear to be in S_B according to the filter. False positives yield elements of S_A that are in fact not in S_B, but, if desired, B can then determine these elements to find $S_A \cap S_B$ exactly. The Bloom filter approach allows $S_A \cap S_B$ to be determined with fewer bits transmitted than A sending the entire set S_A. Reynolds and Vahdat describe how using this approach for set intersection allows for efficient distributed inverted keyword indices for keyword search in an overlay network over a peer-to-peer architecture. When a document is published, the author also selects a set of keywords for the document. Each node in the network is responsible for a set of keywords in the inverted index; hashes of the keyword determine the responsible nodes. To handle conjunctive queries involving multiple nodes, the set intersection methods above are used to reduce the amount of information that needs to be sent to determine the appropriate documents.

6. Applications: Resource Routing

6.1. A Basic Routing Protocol

Before describing specific resource routing protocols in the literature, we provide a general framework that highlights the main idea of resource routing protocols. This general framework was described by Czerwinski et al. as part of their architecture for a resource discovery service [Czerwinski et al. 99].

Suppose that we have a network in the form of a rooted tree, with nodes holding resources. Resource requests starting inside the tree head toward the root. Each node keeps a unified list of resources that it holds or that are reachable through any of its children, as well as individual lists of resources for it and each child. When a node receives a request for a resource, it checks its unified list to make sure it has a way of routing that request to the resource. If it does, it then checks the individual lists to find whether it holds the resource or how to route the request toward the proper node via an appropriate child node; otherwise, it passes the request further up the tree toward the root.

This rather straightforward routing protocol becomes more interesting if the resource lists are represented by Bloom filters. The property that a union of Bloom filters can be obtained by ORing the corresponding individual Bloom filters allows easy creation of unified resource lists. False positives in this situation may cause a routing request to go down an incorrect path. In such a case backtracking up the tree may be necessary, or a slower but safer routing

mechanism may be used as a back-up. Several recent papers utilize a resource routing mechanism of this form.

6.2. Resource Routing on P2P Networks

Rhea and Kubiatowicz [Rhea and Kubiatowicz 02] utilize the ideas in the basic protocol in Section 6.1 to design a probabilistic routing algorithm for peer-to-peer location mechanisms, in conjunction with the OceanStore project [Kubiatowicz et al. 00]. The goal is to ensure that when a requested file has a file replica nearby in the system, it is found and the request is routed efficiently along a shortest path. Such an algorithm can be used in conjunction with a more expensive routing algorithm such as those suggested for specific P2P networks [Druschel and Rowstron 01, Ratnasamy et al. 01, Stoica et al. 01].

Rhea and Kubiatowicz have each node in the network keep an array of Bloom filters for every adjacent edge in the overlay topology. In the array for each edge, there is a Bloom filter for each distance d, up to some maximum value, so that the dth Bloom filter in the array keeps track of files reachable via d hops through the overlay network along that edge. Rhea and Kubiatowicz call this array of Bloom filters an *attenuated Bloom filter*. The attenuated Bloom filter only finds files within d hops, but it is likely to find the shortest path to a file replica if many such paths exist. A more expensive algorithm can be applied if the file cannot be found according to the attenuated Bloom filter or if more than d hops are taken, which suggests that a false positive has occurred. Major challenges in this approach involve keeping the Bloom filters up-to-date without generating too much network traffic.

6.3. Geographic Routing

Hsiao suggests using this type of routing for a geographic routing system for mobile computers [Hsiao 01]. For convenience, suppose that the geographic space is a square region that is recursively subdivided into smaller squares, each one-fourth the area of the previous level. That is, each parent square is broken into four children squares, giving a natural implicit tree hierarchy. If the smallest square subregions have side length 1 and the side length of the original square is k, there will be $\log_2 k + 1$ levels in this recursive structure.

For the geographic routing scheme, each node contains a Bloom filter representing the list of mobile hosts reachable through itself or through its three siblings at each level. Using these filters, a source finds the level corresponding to the smallest geographic region that currently contains it and its desired destination, and then forwards a message to the center of the sibling at that level

that contains the destination node. Intermediate nodes forward the message appropriately, recursing down the implicit tree until the destination is reached.

Distributed hashing has also been proposed as a means of accomplishing geographic routing [Li et al. 00]. So for both P2P network and geographic routing, Bloom filters have been suggested as a possible alternative to distributed hashing that may prove better for systems of intermediate size. Exploring and understanding the tradeoffs between these two techniques would certainly be an interesting area for future work.

7. Applications: Packet Routing

In the area of packet routing, several diverse uses of Bloom filters have been proposed. We examine how Bloom filters can be used to aid early detection of forwarding loops, to find heavy flows for the Stochastic Fair Blue queue management scheme, and to potentially speed up the forwarding of multicast packets.

7.1. Detecting Loops in Icarus

Whitaker and Wetherall suggest using a small Bloom filter in order to avoid forwarding loops in unicast and multicast protocols [Whitaker and Wetherall 02]. Normally packets trapped in a forwarding loop are detected and removed from a network using the IP Time-To-Live field, whereby a counter keeps track of the number of hops that the packet has taken and removes it if the number of hops grows too large. If loops are small, the Time-To-Live field may not prevent substantial unnecessary looping. While such loops are rare in the long-standing protocols guiding most Internet traffic today, the authors suggest that it could be a significant problem for experimental protocols, such as those being suggested for peer-to-peer networks. To avoid this problem, the authors suggest that each packet carry a small Bloom filter in each header, where the Bloom filter is used to keep track of the set of nodes visited. Each node has a corresponding mask that can be ORed into the Bloom filter as it passes; if the filter does not change, there may be a loop.

7.2. Queue Management: Stochastic Fair Blue

Stochastic Fair Blue provides a queue management algorithm that uses a counting Bloom filter to detect overly aggressive or non-responsive flows [Feng et al. 01]. (The idea of using Bloom filters to detect flow behavior arises again in our discussion of applications of Bloom filters to measurement tools in Section 8.1.) Each packet arrival increments k counters of a counting Bloom filter based on, for example, hashes of the source-destination pair, so all packets in a flow hash

to the same counters. When a packet is processed, the corresponding counters are decremented. Each Bloom filter entry has an associated value p_i, used to represent a marking probability associated with that counter. The marking probability associated with a counter goes up by some δ if, when a packet arrives, the number of packets queued in the system corresponding to that counter lies above some threshold; similarly, if, when a packet arrives, there are no packets queued in the system corresponding to that counter, then the marking probability is decreased by δ. The probability that a packet is marked, which denotes congestion to the end hosts, is the minimum of the marking probabilities associated with the k Bloom filter counters after arrival. Flows that are filling a buffer will therefore have higher probabilities of being marked. Flows that are non-responsive to marking will eventually drive the marking probability high; when it is above a certain threshold, the router can limit the flow to a fixed amount of bandwidth or adopt some other rate-limiting policy.

A false positive can occur in this situation if a well-behaved flow happens to hash into k counters that are also hashed into by non-responsive flows. In this case a flow might be punished even though it responds in a proper fashion. One way to mitigate this effect, suggested in [Feng et al. 01], is to change the hash functions periodically, so that if a responsive flow is being punished unfairly the resetting of the hash functions makes it extremely unlikely that it continues to be punished.

7.3. Multicast

When packets are being sent through a multicast tree, the router associates multicast addresses with interface lists. One way to think of this is that each multicast address corresponds to an associated list of interfaces, or connections; if a packet associated with a multicast address comes in on one interface of the list associated with an address, it should be forwarded through all other interfaces on the list.

Grönvall suggests an alternative using Bloom filters [Grönvall 00]. Instead of keeping a list of interfaces for each address, there can be a Bloom filter of addresses associated with each interface. When a packet with a multicast address arrives on one interface, the Bloom filters for all the other interfaces are checked to see if packets with that address should be forwarded along that interface. This avoids the need entirely to store addresses at the router. Parallelization can be used to speed the check of each packet against all interfaces. Handling the removal of an address from an interface is not discussed, but one could imagine using a counting Bloom filter to handle deletions from the Bloom filter accordingly.

8. Applications: Measurement Infrastructure

A growing problem for networks is how to provide a reasonable measurement infrastructure. How many packets from a given flow pass through a router? Has a packet from this source passed through this router recently? The challenge in coping with such questions lies in the tremendous amounts of data being processed, making complete measurement extremely expensive. Because of their succinctness, Bloom filters may be useful for many such problems, as the examples in this section illustrate.

8.1. Recording Heavy Flows

Estan and Varghese present an excellent application of Bloom filters to traffic measurement problems inside of a router, reminiscent of the techniques used in the Stochastic Fair Blue algorithm [Estan and Varghese 02]. (While the authors do not label their data structure a Bloom filter variation, it will be clear that it is from the discussion below.)

The goal is to easily determine heavy flows in a router. Each packet entering is hashed k times into a Bloom filter. Associated with each location in the Bloom filter is a counter that records the number of packet bytes that have passed through the router associated with that location. The counter is incremented by the number of bytes in the packet. If the minimum counter associated with a packet is above a certain threshold, the corresponding flow is placed on a list of heavy flows. Heavy flows can thereby be detected with a small amount of space and a small number of operations per packet.

A false positive in this situation corresponds to a light flow that happens to hash into k locations that are also hashed into by heavy flows, or to a light flow that happens to hash into locations hit by several other light flows. Estan and Varghese introduce the idea of a *conservative update*, an interesting variation that reduces the false positive rate significantly for real data. When updating a counter upon a packet arrival, it is clear that the number of previous bytes associated with the flow of that packet is at most the minimum over its k counters. Call this M_k. If the new packet has B bytes, the number of bytes associated with this flow is at most $M_k + B$. So the updated value for each of the k counters should be the maximum of its current value and $M_k + B$. Instead of adding B to each counter, conservative update only changes the values of the counters to reflect the most possible bytes associated with the flow, as shown in the example in Figure 2. This reduces the probability that several light flows hashing to the same location can raise the counter value at this location over the threshold.

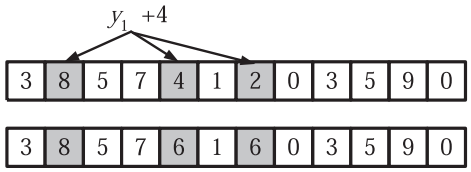

Figure 2. An example of conservative update. This flow can only have been responsible for 2 previous bytes, so when it introduces 4 new bytes, counters should increase only to 6.

8.2. IP Traceback

If one wanted to trace the route that a packet took in a network, one way of doing it would be to have each router in the network record every packet that it forwards. Then each router could be queried to determine whether it forwarded the given packet, allowing the route of the packet to be traced backward from its destination. Such a scheme would allow malicious packets to be traced back along uncorrupted routers in order to find their source.

Snoeren et al. suggest this approach, with the addition of using Bloom filters in order to reduce the amount of information that needs to be stored in order to summarize the set of packets seen, as part of their Source Path Isolation Engine (SPIE) [Snoeren et al. 01]. A false positive in this setting means that a router mistakenly identifies a packet as having been seen. When attempting to trace back the reverse path of a packet, a false positive would lead to a branch, giving multiple possible paths. A low false positive rate would keep the branching small and hence the number of possible paths small as well. Of course, to make such a scheme practical, the authors give careful consideration to how much information to store and when to discard stale information.

9. Recent Work

Since the initial version of this survey [Broder and Mitzenmacher 02], there has been significant additional work on Bloom filters. We briefly highlight some relevant theoretical work that we expect will be useful in future network applications.

Cohen and Matias introduce *spectral Bloom filters* [Cohen and Matias 03], which, like the work of Estan and Varghese of Section 8.1, extend the basic Bloom filter construction so that it can handle multi-sets. One of the innovations in their work is to use a second filter to handle elements that have a unique minimum counter in the filter to improve the accuracy of the resulting estimates. Cormode

and Muthukrishnan devise the *count-min sketch*, a variation on the Bloom filter designed to handle several problems on data streams, again similar to the work of Estan and Varghese [Cormode and Muthukrishnan 04]. They are able to provide theoretical guarantees while using only pairwise independent hash functions; this is a significant advance, since pairwise independent hash functions are generally easy to implement and quite efficient in practice. Another paper that tackles similar problems introduces the *space-code Bloom filter*, which utilizes multiple Bloom filters and maximum likelihood estimation in order to approximate multi-sets [Kumar et al. 03]. The goal motivating this variation is to obtain accurate estimates of packet counts for all flows on a network router, not only the large flows. Chazelle, Kilian, Rubinfeld, and Tal [Chazelle et al. 04] introduce what they call a *Bloomier filter*, which extends the Bloom filter to handle situations where each element of a set S is associated with a function value (from a discrete and finite set of values); the function value of all other elements of the universe are assumed to be undefined. The Bloomier filter provides the appropriate function value for any element in S and returns a value corresponding to undefined for any element not in S, except that for elements not in S, there is some probability of a false positive, whereby the Bloomier filter may return an incorrect function value. This work amply demonstrates that there remain many ways of extending Bloom filters and interesting corresponding theoretical problems.

10. Conclusion

A Bloom filter is a simple space-efficient representation of a set or a list that handles membership queries. As we have seen in this survey, there are numerous networking problems where such a data structure is required. Especially when space is an issue, a Bloom filter may be an excellent alternative to keeping an explicit list. The drawback of using a Bloom filter is that it allows false positives. Their effect must be carefully considered for each specific application to determine whether the impact of false positives is acceptable. This leads us back to:

The Bloom filter principle: *Wherever a list or set is used, and space is at a premium, consider using a Bloom filter if the effect of false positives can be mitigated.*

There seems to be plenty of room to develop variants or extensions of Bloom filters for specific applications. For example, we have seen that the counting Bloom filter allows for approximate representations of multi-sets or dynamic sets that change over time through both insertions and deletions. Bloom filters

are now starting to receive significant attention from the algorithmic community, and while there have been a number of recent results, there may well be further improvements to be found.

We expect that the recent burst of applications is really just the beginning. Because of their simplicity and power, we believe that Bloom filters will continue to be used in modern network systems in new and interesting ways.

Acknowledgments. A preliminary version of this survey appeared at the 2002 Allerton Conference [Broder and Mitzenmacher 02]. The authors would like to thank our many colleagues and the anonymous referees that pointed us to related work and helped us improve the presentation of the material. The second author was supported in part by NSF grants CCR-9983832, CCR-0118701, CCR-0121154, and an Alfred P. Sloan Research Fellowship.

References

[Bloom 70] B. Bloom. "Space/Time Tradeoffs in Hash Coding with Allowable Errors." *Communications of the ACM* 13:7 (1970), 422–426.

[Broder and Mitzenmacher 02] A. Z. Broder and M. Mitzenmacher. "Network Applications of Bloom Filters: A Survey." In *Proceedings of the Fortieth Annual Allerton Conference on Communication, Control, and Computing*. CD-ROM. Coordinated Science Laboratory and the Department of Electrical and Computer Engineering of the University of Illinois at Urbana-Champaign, 2002.

[Bratbergsengen 84] K. Bratbergsengen. "Hashing Methods and Relational Algebra Operations." In *Proceedings of the Tenth International Conference on Very Large Databases*, pp. 323–333. San Francisco: Morgan Kaufmann, 1984.

[Byers et al. 02a] J. Byers, J. Considine, M. Mitzenmacher, and S. Rost. "Informed Content Delivery over Adaptive Overlay Networks." *ACM SIGCOMM Computer Communication Review (Proceedings of the 2002 SIGCOMM Conference)* 32:4 (2002), 47–60.

[Byers et al. 02b] J. Byers, J. Considine, and M. Mitzenmacher. "Fast Approximate Reconciliation of Set Differences." Boston University Technical Report 2002-019, July 2002.

[Chazelle et al. 04] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. "The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables." In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 30–39. Philadelphia: SIAM, 2004.

[Cohen and Matias 03] S. Cohen and Y. Matias. "Spectral Bloom Filters." In *Proceedings of the 2003 ACM SIGMOD International Conference on the Management of Data*, pp. 241–252. New York: ACM Press, 2003.

[Cormode and Muthukrishnan 04] G. Cormode and S. Muthukrishnan. "An Improved Data Stream Summary: The Count-Min Sketch and its Applications." In *LATIN*

2004: Theoretical Informatics, edited by M. Farach-Colton, pp. 29–38. New York: ACM Press, 2004.

[Cuena-Acuna et al. 02] F. M. Cuena-Acuna, C. Peery, R. P. Martin, and T. D. Nguyen. "PlanetP: Using Gossiping to Build Content Addressable Peer-to-Peer Information Sharing Communities." Rutgers Technical Report DCS-TR-487, 2002.

[Czerwinski et al. 99] S. Czerwinski, B. Y. Zhao, T. Hodes, A. D. Joseph, and R. Katz. "An Architecture for a Secure Service Discovery Service." In *Proceedings of the Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '99)*, pp. 24–35. New York: ACM Press, 1999.

[Druschel and Rowstron 01] P. Druschel and A. Rowstron. "Storage Management and Caching in PAST, A Large-Scale, Persistent Peer-to-Peer Storage Utility." In *Proceedings of the Eighteenth ACM Symposium on Operations Systems Principles*, pp. 188–201. New York: ACM Press, 2001.

[Dubhasi and Ranjan 98] D. Dubhasi and D. Ranjan. "Balls and Bins: A Study in Negative Dependence." *Random Structures and Algorithms* 13:2 (1998), 99–124.

[Estan and Varghese 02] C. Estan and G. Varghese. "New Directions in Traffic Measurement and Accounting." *ACM SIGCOMM Computer Communication Review (Proceedings of the 2002 SIGCOMM Conference)* 32:4 (2002), 323–336.

[Fan et al. 00] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol." *IEEE/ACM Transactions on Networking* 8:3 (2000), 281–293.

[Feng et al. 01] W.-C. Feng, K. G. Shin, D. Kandlur, and D. Saha. "Stochastic Fair Blue: A Queue Management Algorithm for Enforcing Fairness." In *Proceedings of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM-01)*, Volume 3, pp. 1520–1529. Los Alamitos, CA: IEEE Computer Society, 2001.

[Gremilion 82] L. L. Gremilion. "Designing a Bloom Filter for Differential File Access." *Communications of the ACM* 25 (1982), 600–604.

[Grönvall 00] B. Grönvall. "Scalable Multicast Forwarding." Available from World Wide Web (www.acm.org/sigcomm/ccr/archive/2002/jan02/CCR-SC01-Posters/BjornGronvall.ps), 2002.

[Hsiao 01] P. Hsiao. "Geographical Region Summary Service for Geographical Routing." *Mobile Computing and Communications Review* 5:4 (2001), 25–39.

[Kumar et al. 03] A. Kumar, J. Xu, L. Li, and J. Wang. "Space-Code Bloom Filter for Efficient Traffic Flow Measurement." In *Proceedings of the 2003 ACM SIGCOMM Conference on Internet Measurement*, pp. 167–172. New York: ACM Press, 2003.

[Kubiatowicz et al. 00] J. Kubiatowicz, D. Bindel, P. Eaton, Y. Chen, D. Geels, R. Gummadi, S. Rhea, W. Weimer, C. Wells, H. Weatherspoon, and B. Zhao. "OceanStore: An Architecture for Global-Scale Persistent Storage." *ACM SIGPLAN Notices* 35:11 (2000), 190–201.

[Ledlie et al. 02] J. Ledlie, J. Taylor, L. Serban, and M. Seltzer. "Self-Organization in Peer-to-Peer Systems." In *Tenth ACM SIGOPS European Workshop*. Available from World Wide Web (http://www.stanford.edu/∼candea/portals/SIGOPS-2002/), 2002.

[Li et al. 00] J. Li, J. Jannotti, D. De Couto, D. Karger, and R. Morris. "A Scalable Location Service for Geographic Ad-Hoc Routing." In *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking (MobiCom)*, pp. 120–130. New York: ACM Press, 2000.

[Mackett and Lohman 86] L. F. Mackett and G. M. Lohman. "R* Optimizer Validation and Performance Evaluation for Distributed Queries." In *Proceedings of the Twelfth International Conference on Very Large Databases*, pp. 149–159. San Francisco, CA: Morgan Kaufmann, 1986.

[Manber and Wu 94] U. Manber and S. Wu. "An Algorithm for Approximate Membership Checking with Application to Password Security." *Information Processing Letters* 50 (1994), 191–197.

[Marais and Bharat 97] H. Marais and K. Bharat. "Supporting Cooperative and Personal Surfing with a Desktop Assistant." In *Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology*, pp. 129–138. New York: ACM Press, 1997.

[McIlroy 82] M. D. McIlroy. "Development of a Spelling List." *IEEE Transactions on Communications* 30:1 (1982), 91–99.

[Mitzenmacher 02] M. Mitzenmacher. "Compressed Bloom Filters." *IEEE/ACM Transactions on Networking* 10:5 (2002), 604–612.

[Mullin 83] J. K. Mullin. "A Second Look at Bloom Filters." *Communications of the ACM* 26:8 (1983), 570–571.

[Mullin and Margoliash 90] J. K. Mullin and D. J. Margoliash. "A Tale of Three Spelling Checkers." *Software – Practice and Experience* 20:6 (1990), 625–630.

[Mullin 90] J. K. Mullin. "Optimal Semijoins for Distributed Database Systems." *IEEE Transactions on Software Engineering* 16:5 (1990), 558.

[Mullin 93] J. K. Mullin. "Estimating the Size of a Relational Join." *Information Systems* 18:3 (1993), 189–196.

[Ramakrishna 89] M. V. Ramakrishna. "Practical Performance of Bloom Filters and Parallel Free-Text Searching." *Communications of the ACM* 32:10 (1989), 1237–1239.

[Ratnasamy et al. 01] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. "A Scalable Content-Addressable Network." *ACM SIGCOMM Computer Communication Review (Proceedings of the 2001 SIGCOMM Conference)* 31:4 (2001), 161–172.

[Reynolds and Vahdat 03] P. Reynolds and A. Vahdat. "Efficient Peer-to-Peer Keyword Searching." In *Middleware 2003: ACM/IFIP/USENIX International Middleware Conference, Rio de Janeiro, Brazil, June 16–20, 2003, Proceedings*, Lecture Notes in Computer Science 2672, pp. 21–40. New York: Springer, 2003.

[Rhea and Kubiatowicz 02] S. C. Rhea and J. Kubiatowicz. "Probabilistic Location and Routing." In *Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, Volume 3, pp. 1248–1257. Los Alamitos, CA: IEEE Computer Society, 2002.

[Rousskov and Wessels 98] A. Rousskov and D. Wessels. "Cache Digests." *Computer Networks and ISDN Systems* 30:22-23 (1998), 2155–2168.

[Snoeren et al. 01] A. C. Snoeren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchakountio, S. T. Kent, and W. T. Strayer. "Hash-Based IP Traceback." *ACM SIGCOMM Computer Communication Review (Proceedings of the 2001 SIGCOMM Conference)* 31:4 (2001), 3–14.

[Spafford 92] E. H. Spafford. "Opus: Preventing Weak Password Choices." *Computer and Security* 11 (1992), 273–278.

[Stoica et al. 01] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications." *ACM SIGCOMM Computer Communication Review (Proceedings of the ACM 2001 SIGCOMM Conference)* 31:4 (2001), 149–160.

[Valdurez and Gardarin 84] P. Valdurez and G. Gardarin. "Join and Semijoin Algorithms for a Multiprocessor Database Machine." *ACM Transactions on Database Systems* 9:1 (1984), 133–161.

[Whitaker and Wetherall 02] A. Whitaker and D. Wetherall. "Forwarding without Loops in Icarus." In *Proceedings of the Fifth IEEE Conference on Open Architectures and Network Programming (OPENARCH)*, pp. 63–75. Los Alamitos, CA: IEEE Computer Society, 2002.

Andrei Broder, IBM Research Division, Hawthorne, NY 10532 (abroder@us.ibm.com)

Michael Mitzenmacher, Harvard University, Division of Engineering and Applied Sciences, Cambridge, MA 02138 (michaelm@eecs.harvard.edu)

Received April 13, 2004; accepted May 10, 2004.