Network Applications of Bloom Filters: A Survey*

Andrei Broder† Michael Mitzenmacher‡

Abstract

A Bloom filter is an ingenious randomized data-structure for concisely representing a set in order to support approximate membership queries. The space efficiency is achieved at the cost of a small probability of false positives. It was invented by Burton Bloom in 1970 for the purpose of spell checking and for many years it was seldom mentioned in other contexts, except for database optimization. Nevertheless, Bloom's beautiful approach has seen a sudden resurgence in a variety of large-scale network applications such as shared web caches, query routing, and replica location. This survey presents a plethora of recent uses of this old data structure, its modern variants, and the mathematical basis behind them, with the aim of making these ideas available to a wider community and the hope of inspiring new applications.

1 Introduction

A Bloom filter is a simple space-efficient randomized data structure for representing a set in order to support membership queries. The space efficiency is achieved at the cost of a small probability of false positives, but often this is a convenient trade-off. Although Bloom filters were invented in the 1970's [1] and have been heavily used in database applications (see e.g. [20, 15]), they have only recently received widespread attention in the networking literature.

This survey presents a plethora of recent uses of Bloom filters in a variety of network contexts, with the aim of making these ideas available to a wider community and the hope of inspiring new applications. We first describe the mathematics behind Bloom filters, 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. If a specific application whets your curiosity, we encourage you to read the full papers.

The theme unifying these diverse applications is that a Bloom filters offers a succinct way of representing a set or 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 a consideration, a Bloom filter should be considered. When using a Bloom filter, consider the potential effects of false positives.

*This is a shortened, preliminary version of a longer survey, that will made available at http://www.eecs.harvard.edu/~michaelm

†IBM Research Division. E-mail: abroder@ibm.us.com.

‡Harvard University, Div. of Engineering and Applied Sciences. Supported in part by NSF grants CCR-9983832, CCR-0118701, CCR-0121154, and an Alfred P. Sloan Research Fellowship. E-mail: michaelm@eecs.harvard.edu.

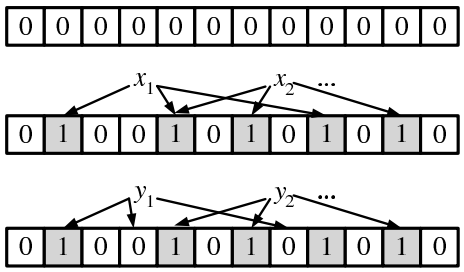

Figure 1: An example of a Bloom filter. The filter begins as an array of all 0's. Each item x_i in the set S is hashed k times, with each hash yielding a bit location; these bits are set to 1. To check if an element y is in the set, hash it k times and check the corresponding bits. The element y_1 cannot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

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 $\{1, \ldots, m\}$. We make the natural assumption that these hash functions map each item in the universe to a random number uniform over the range $\{1, \ldots, m\}$ for mathematical convenience. (In practice, reasonable hash functions appear to behave adequately, e.g. [21].) 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 y 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.

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

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

We let $p = \mathrm{e}^{-kn/m}$. The probability of a false positive is then

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

We let $f = \left(1 - \mathrm{e}^{-kn/m}\right)^k = (1-p)^k$. Note that we use the asymptotic approximations from now on for convenience.

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

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

the probability of a false positive is actually always slightly higher with this division. Since the difference is small, this approach may be still be useful for implementation reasons; for example, dividing the bits among the hash functions may make parallelization of array accesses easier.

Suppose 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 equals $\exp(k \ln(1 - \mathrm{e}^{-kn/m}))$. 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

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

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

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

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 $(1/2)^k = (0.6185)^{m/n}$. In practice, of course, k must be an integer, and smaller k might be preferred since they reduce the amount of computation necessary.

2.2 Hashing vs. Bloom filters

Another natural way to represent a set is to use hashing. Each item of the set can be 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 obtain 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 hashing.) Bloom filters yield a constant false positive probability even if a constant number of bits are used 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 interesting; using hashing yields an asymptotically vanishing probability of error with only $\Theta(\log n)$ bits per element. Bloom filters have therefore received little attention in the theoretical community. In contrast, for practical applications the price of a constant false positive probability may well be worthwhile to reduce the necessary space.

2.3 Standard Bloom filter tricks

The simple structure of Bloom filters makes certain operations very easy to implement. For example, suppose one has two Bloom filters representing sets S_1 and S_2 with the same number of bits and using the same number of 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. Suppose the size of the filter is a power of 2. If one wants to half the size of the filter, just OR the first and second halves together. When hashing, the high order bit can be masked.

2.4 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, [8] introduces the idea of a *counting Bloom filter*.[1] In a counting Bloom filter, each entry in the Bloom filter is not a single bit but instead 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. Analysis from [8] (which will appear in the full version of the survey) reveals that 4 bits per counter should suffice for most applications.

2.5 Compressed Bloom filters

In recent work, Mitzenmacher addresses the following question [19]. 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 above, then $p = 1/2$. Under our assumption of good random hash functions, the bit array is essentially a random string of m 0's and 1's, with each entry being 0 or 1 with probability $1/2$. It would therefore seem that compression cannot gain when sending Bloom filters.

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 by using a larger, but sparser, Bloom filter can yield improved false positive rates with a smaller number of transmitted bits.

3 Historical Applications

Bloom filters were used in early UNIX spell-checkers [18]. Rather than store and search a dictionary, a Bloom filter representation of the dictionary was stored. 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 by Spafford as a means of succinctly storing a dictionary of unsuitable passwords for security purposes [27]. Manber and Wu describe a simple way to extend the technique so that passwords that are within edit distance 1 of a dictionary word are also not allowed [16]. 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.

Bloom filters have also been used in databases, see e.g. [15, 20]. One use is to speed up semijoin operations. In a distributed database, one database may wish to send another a list of all cities where the cost of living is greater than 50,000 dollars, so that the other database can determine all employees that live in such cities. Instead of sending a list of cities, a Bloom filter can be sent. The list of employee/city pairs can be sent back to the first database to remove false positives.

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 [8]. 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 Internet.

[1] The name counting Bloom filter for this data structure was introduced in [19].

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 using 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 [25].

Since cache contents are changing frequently, [8] 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 [19], can yield a further reduction in the number of bits transmitted.

5 Applications: P2P/Overlay Networks

Peer-to-peer applications are a natural place to use Bloom filters, as collaborating peers may need to send each other lists of URLs, packets, or object identifiers. As an example, an early peer-to-peer application of Bloom filters is due to Marais and Bharat [17] in the context of a desktop web browsing assistant called *Vistabar*. Cooperative users of Vistabar store annotations and comments about the web pages 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 periodically downloads a Bloom filter corresponding to all annotated URLs.

5.1 Moderate-sized P2P networks

Many constructions for peer-to-peer networks are based on distributed hash tables in order to locate objects [6, 22, 28]. Distributed hash tables are particularly useful for large-scale scalability and for coping with settings 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 [4]; the filters actually store keywords associated with documents instead of object IDs. Implementation challenges include how frequently filters need to be updated.

In [13], 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 me 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 [3] 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. B would like to send 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 sends 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 size of the difference is small [3, 2]. This work demonstrates that Bloom filters can also be useful as subroutines inside of more complex data structures.

The application [3] targets is 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, receiving encoded packets from other peers as well as the source, effectively increasing the download rate. The problem of determining what encoded packets peer B needs that peer A has is simply the problem of determining $S_B - S_A$. Since the content is redundantly encoded, obtaining a large fraction of $S_B - S_A$ rather than the entire set is sufficient for this application.

5.3 Set Intersection for Keyword Searches

Reynolds and Vahdat use Bloom filters in a similar fashion as [3], except their goal is to find the set intersection instead of the set difference [23]. Their approach is essentially the same as for database semijoins. Peer B can send a Bloom filter representing S_B to A; 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 by having A sending the entire set S_A. Reynolds and Vahdat describe how 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.

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 [5].

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 checks the individual lists to find how to route the request toward the proper 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 [24] utilize the ideas in the basic protocol above to design a probabilistic routing algorithm for peer-to-peer location mechanisms, in conjunction with the OceanStore project

[12]. 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 [6, 22, 28].

Rhea and Kubiatowicz have each node in the network keep an array of Bloom filters for every edge in the overlay topology. 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 along an edge via d hops through the overlay network. 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 paths exist. A more expensive algorithm can be applied if the file cannot be found using the attenuated Bloom filter or if more than d hops are taken, which suggests 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 [11]. For convenience, suppose that the geographic space is a square region that is recursively subdivided into smaller squares, each one-fourth the size 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 size 1 and the size 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 contains it and the destination, and then forwards a message to the center of the region corresponding to the sibling that the destination node currently resides in. 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 accomplished geographic routing [14]. 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 [29]. 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 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 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. False positives may lead to packets incorrectly

being dropped because of an assumed loop. The authors discuss ways to limits the negative effects of false positives in this context.

7.2 Queue Management: Stochastic Fair Blue

Stochastic Fair Blue provides a queue management algorithm that uses a Bloom filter to detect overly aggressive or non-responsive flows [9]. The idea of using a Bloom filter to detect flow behavior arises again in our discussion of applications of Bloom filters to measurement tools.

Each packet is hashed into k bits in a Bloom filter based on for example the source-destination pair, so all packets in a flow hash to the same bits. Each Bloom filter entry has an associated value p_i, used to represent a marking probability associated with that bit. The marking probability associated with a bit goes up by some value δ if, when a packet arrives, the number of packets queued in the system corresponding to that bit lies above some threshold; similarly, if when a packet arrives there are no packets queued in the system corresponding to that bit, then the marking probability is decreased by δ. The probability that a packet is marked, which will denote congestion to the end hosts, is the minimum of the marking probabilities associated with the k Bloom filter bits 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 in this situation leads to the misclassification of a well-behaved flow. In this case a flow might be punished even though it responds to congestion appropriately. One way to mitigate this effect suggested in [9] 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 [10]. Instead of keeping a list of interfaces for each address, there can be a Bloom filter of addresses associated with each interface. This avoids the need to store addresses at the router entirely. 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.

False positives in this setting lead to some packets being forwarded incorrectly. These packets will eventually be discarded, either at the next router hop or further down the line, so if the false positive probability is small the effect may not be significant.

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 below 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 [7]. (While

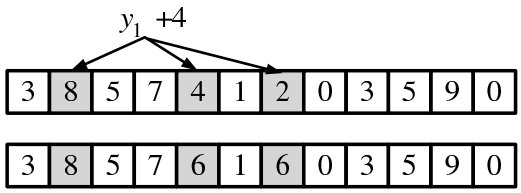

Figure 2: An example of conservative update. This flow can only have been responsible for 2 previous bytes, so when it introduces 4 new bytes, counters should increase only to 6.

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 entering packet 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. All heavy flows, however, are detected.

Estan and Varghese also 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 counter 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 over the threshold.

8.2 IP Traceback

If one wanted to trace the route 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 summarize the set of packets seen, as part of their Source Path Isolation Engine (SPIE) [26]. 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 branching, 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 Conclusion

A Bloom filter is a space-efficient representation of a set or a list that handles membership queries. As we have seen in this survey, there are numerous examples where one would like to use a list

in a network. 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 introduces false positives. The effect of a false positive must be carefully considered for each specific application to determine whether the impact of false positives is acceptable. This leads us to:

The Bloom filter principle: Wherever a list or set is used, and space is a consideration, a Bloom filter should be considered. When using a Bloom filter, consider the potential effects of false positives.

There seems to be a great deal 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 allows one to track sets that change over time through insertions and deletions. Since Bloom filters have received comparatively little attention from the algorithmic community, there may be a number of improvements to be found.

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

References

[1] B. Bloom. Space/time tradeoffs in in hash coding with allowable errors. *CACM*, 13(7):422-426, 1970.

[2] J. Byers, J. Considine, and M. Mitzenmacher. Fast approximate reconciliation of set differences. Boston University Computer Science Technical Report 2002-019.

[3] J. Byers, J. Considine, M. Mitzenmacher, and S. Rost. Informed content delivery over adaptive overlay networks. In *Proc. of the ACM SIGCOMM 2002 Conference (SIGCOMM-02)*, vol. 32:4 of *Computer Communication Review*, pages 47–60, October 2002.

[4] 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.

[5] S. Czerwinski, B. Y. Zhao, T. Hodes, A. D. Joseph, and R. Katz. An architecture for a secure service discovery service. In *Proc. of MobiCom-99*, pages 24–35, N.Y., August 1999.

[6] P. Druschel and A. Rowstron. PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the Eighth Workshop on Hot Topics in Operations Systems*, May 2001.

[7] C. Estan and G. Varghese. New directions in traffic measurement and accounting. In *Proceedings of the ACM SIGCOMM 2002 Conference (SIGCOMM-02)*, volume 32:4 of *Computer Communication Review*, pages 323–336, October 2002.

[8] 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):281–293, 2000.

[9] 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)*, pages 1520–1529, Los Alamitos, CA, Apr. 22–26 2001. IEEE Computer Society.

[10] B. Grönvall. Scalable multicast forwarding. Available at `www.acm.org/sigcomm/ccr/archive/2002/jan02/CCR-SC01-Posters/BjornGronvall.ps`.

[11] P. Hsiao. Geographical region summary service for geographical routing. *Mobile Computing and Communications Review*, 5(4)25–39, October 2001.

[12] 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):190–201, November 2000.

[13] J. Ledlie, J. Taylor, L. Serban, M. Seltzer. Self-organization in peer-to-peer systems. In *Proceedings of the 10th European SIGOPS Workshop*, September 2002.

[14] J. Li and J. Jannotti and D. De Couto and D. Karger and R. Morris. A scalable location service for geographic ad-hoc routing. In *Proceedings of MobiCom 2000*, pages 120–130, August 2000.

[15] Z. Li and K. A. Ross. Perf join: An alternative to two-way semijoin and bloomjoin. In *CIKM '95, Proceedings of the 1995 International Conference on Information and Knowledge Management*, pages 137–144, November 1995.

[16] U. Manber and S. Wu. An algorithm for approximate membership checking with application to password security. *Information Processing Letters*, 50:191-197, 1994.

[17] H. Marais and K. Bharat. Supporting cooperative and personal surfing with a desktop assistant. In *ACM Symposium on User Interface Software and Technology*, pages 129–138, 1997.

[18] M. D. McIlroy. Development of a spelling list. *IEEE Transactions on Communications*, 30(1):91–99, January 1982.

[19] M. Mitzenmacher. Compressed bloom filters. In *Proceedings of the 20th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 144–150, August 2001.

[20] J. K. Mullin. Optimal semijoins for distributed database systems. *IEEE Transactions on Software Engineering*, 16(5):558, May 1990.

[21] M. V. Ramakrishna. Practical performance of Bloom filters and parallel free-text searching. *Communications of the ACM*, 32(10):1237-1239, October 1989.

[22] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM 2001 Conference (SIGCOMM-01)*, volume 31:4 of *Computer Communication Review*, pages 161–172, August 2001.

[23] P. Reynolds and A. Vahdat. Efficient peer-to-peer keyword searching. Unpublished manuscript.

[24] S. C. Rhea and J. Kubiatowicz. In *Proc. of INFOCOM-02*, June 2002.

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

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

[27] E. H. Spafford. Opus: Preventing weak password choices. *Computer and Security*, 11:273-278, May 1992.

[28] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of the ACM SIGCOMM 2001 Conference (SIGCOMM-01)*, volume 31:4 of *Computer Communication Review*, pages 149–160, August 2001.

[29] A. Whitaker and D. Wetherall Forwarding without loops in Icarus. In *Proceedings of the Fifth OPENARCH*, pages 63–75, June 2002.