Space Efficient storage of email to address security questions

Sendmail:

Sendmail is an *open-source* version of the most popular Mail Transfer Agent (MTA). Sendmail is a general purpose internetwork_email routing facility that supports many kinds of mail-transfer and -delivery methods, including the Simple Mail Transfer Protocol (SMTP) used for email transport over the Internet. Sendmail is a well-known project of the free and open source software and UNIX communities, and has spread both as free software and proprietary software.

Sendmail supports a variety of mail transfer protocols, including SMTP, ESMTP, DECnet's Mail-11, HylaFax, QuickPage and UUCP. Additionally, Sendmail v8.12 introduced support for *milters* - external mail filtering programs that can participate in each step of the SMTP conversation.

The sendmail program plays a variety of roles, all critical to the proper flow of electronic mail. It listens to the network for incoming mail, transports mail messages to other machines, and hands local mail to a local program for local delivery. It can append mail to files and pipe mail through other programs. It can queue mail for later delivery and understand the aliasing of one recipient name to another.

The other major functionality enhancement for Sendmail is a third party mail filter API. This API will allow system administrators and third party companies to provide message filtering using hooks in the sendmail code. This new plug-in architecture will allow for better spam and virus monitoring as well as give administrators the ability to accept, reject, discard, modify, or archive messages.

Milters

Milter (portmanteau for *mail filter*) is an extension to the widely used open source mail transfer agents (MTA) Sendmail and Postfix. It allows administrators to add mail filters for filtering spam or viruses very efficiently in the mail-processing chain. In the language of the art, "milter" refers to the protocol and API implementing the service, while "a milter" has come to refer to a filter application that uses milter to provide service.

A Milter is a multithreaded program that is called by sendmail whenever mail is received using SMTP. Because sendmail forks a copy of itself for each delivery, there is no part of the sendmail system that is aware of the big picture. A well-designed Milter fulfills a central role in such an arrangement because it can view the big picture. Such a Milter can be a very powerful companion to sendmail.

The sendmail program can support multiple Milter programs and can call each in a predefined order. Also, beginning with sendmail V8.13, Milters can be called in a per-daemon order. For example, the daemon bound to the external network interface might prefer one list of Milters, whereas the daemon bound to the internal network interface might prefer another list.

The role of a Milter depends on two points of view:

1) From the point of view of a single sendmail process, there are (possibly) many Milters that it can call in sequence.

2) From the point of view of a single Milter, it receives individual envelopes (messages) in a well-defined order from possibly many simultaneous connections from multiple sendmail processes.

The Milter API can be used on any OS that supports threads and it was created with the following application goals in mind:

- Enhanced Safety and Security
- Enhanced Reliability
- Improved Simplicity
- Improved Performance

The Milter API provides an interface for third-party software to validate and modify messages as they pass through the mail transport system. The MTA configuration file specifies which filters are to be applied, and in what order, allowing an administrator to combine multiple independently-developed filters

Bloom Filters

Bloom filters are compact data structures for probabilistic representation of a set in order to support membership queries (i.e., queries that ask: "Is element X in set Y?").

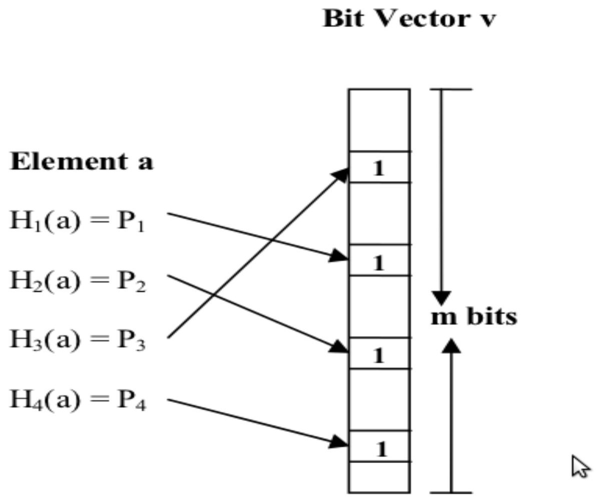


Fig. 2. A Bloom Filter with 4 Hash Functions

This compact representation is the payoff for allowing a small rate of false positives in membership queries; that is, queries might incorrectly recognize an element as member of the set. Since their introduction, Bloom filters have seen various uses such as web cache sharing, query filtering and routing, compact representation of a differential file [5] and free text searching. Consider a set A = {a1, a2... an} of n elements (also called keys). The idea is to allocate a vector v of m bits, initially all set to 0, and then choose k independent hash functions, h1, h2, ... , hk, each with range {1,..., m}. For each element a $\in$ A, the bits at positions h1(a), h2(a), . . . , hk(a) in v are set to 1. A particular bit might be set to 1 multiple times. Given a query for b, we check the bits at positions h1(b), h2(b), . . . , hk(b). If any of them is 0, then certainly b is not in the set A. Otherwise we conjecture that b is in the set although there is a certain probability that we are wrong. This is called a "false positive". The parameters k and m should be chosen such that the probability of a false positive (and hence a false hit) is acceptable. Let BF be the Bloom filter for a set A, we denote the query for b, match (BF, b).

To support updates of the set A we maintain for each location l in the bit array a count c(l) of the number of times that the bit is set to 1 (that is, the number of elements that hashed to l under any of the hash functions). All counts are initially 0. When a key a is inserted or deleted, the counts c(h1(a)), c(h2(a)), . . . , c(hk(a)) are incremented or decremented accordingly. When a count changes from 0 to 1, the corresponding bit is turned on. When a count changes from 1 to 0 the corresponding bit is turned off. In practice, allocating four bits per count is sufficient.

The probability of false positives depends on the number of hash functions we use k, the number of elements we index n, and the size of the Bloom filter m. The formula that gives this probability is: P = (1 – e-kn/m)*k.

The advantage of using Bloom filters is not only the space savings but also the speed of querying. It takes only a short constant time to query the Bloom filter for any packet.

Constructing Bloom Filters

Consider a set $A = \{a_1, a_2, ..., a_n\}$ of n elements. Bloom filters describe membership information of *A* using a bit vector *V* of length *m*. For this, *k* hash functions, $h_1, h_2, ..., h_k$ with $h_i : X \rightarrow \{1..m\}$, are used as described below:

The following procedure builds an *m* bits Bloom filter, corresponding to a set A and using $h_1, h_2, ..., h_k$ hash functions:

```
Procedure BloomFilter(set A, hash_functions, integer m) returns filter
        filter = allocate $m$ bits initialized to 0
        foreach $a_i$ in $A$:
                foreach hash function $h_j$:
                        filter[$h_j(a_i)$] = 1
                end foreach
        end foreach
        return filter
```

Therefore, if a_i is member of a set *A,* in the resulting Bloom filter *V* all bits obtained corresponding to the hashed values of a_i are set to 1. Testing for membership of an element *elm* is equivalent to testing that all corresponding bits of *V* are set:

```
Procedure MembershipTest (elm, filter, hash_functions) returns yes/no
    foreach hash function $h_j$:
        if filter[$h_j(elm)$] != 1 return No
    end foreach
    return Yes
```

Nice features: Filters can be built incrementally: as new elements are added to a set the corresponding positions are computed through the hash functions and bits are set in the filter. Moreover, the filter expressing the reunion of two sets is simply computed as the bit-wise OR applied over the two corresponding Bloom filters.

Proposed system

Today virus and malicious programs are travelling through emails and that affects the computers. To track who sent the malicious mails and who are the victims we have to store entire emails including header, but this creates a high usage of memory. So we want to implement a space efficient storage of emails to address the questions like

1) Whether any mail with some content has passed through this mail server
2) Whether any mail with particular email-id is received by mail server.
3) Whether any mail is sent to particular email-id through this mail server

The system we propose is based on a novel data structure called a Hierarchical Bloom Filter (HBF). An HBF creates compact digests of messages and provides probabilistic answers to member-ship queries.

HIERARCHICAL BLOOM FILTERS

A naive method to store emails that consumes a small amount of storage and also provides some privacy guarantees is to simply store hashes of emails instead of the actual emails. This effectively reduces the amount of data to be stored. Using a standard Bloom filter with k hash functions, we can further reduce this space at the cost of a small false positive rate. For a specific space usage of m bits, n strings (packets) inserted into the Bloom filter, the optimum value for F P is achieved for $k = \log_2(m/n)$ and $FP \approx 0.6185m/n$. So, for example, storage per email can be reduced from 20 bytes to 21 bits at a false positive rate of $4.27\times10-5$. Compared to simple hashes, the only advantage of using standard Bloom filters is the space saving.

Unfortunately the approaches above restrict the queries to the whole email. Attributing part of email is more useful and a simple approach to support queries on part is to hash blocks of the email instead.

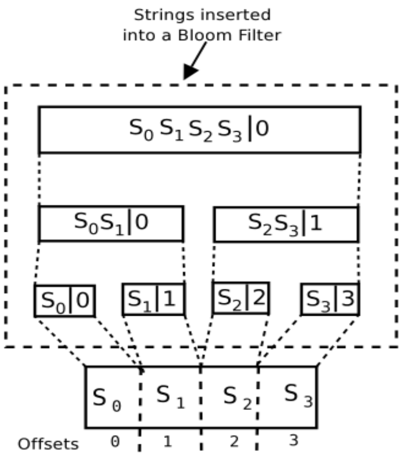


Figure 2: Inserting string "$S_0S_1S_2S_3$" into a Hierarchical Bloom Filter.

An email is inserted into the hierarchy from bottom-up. An email of length p is broken into p/s blocks which are inserted into the HBF at level 0. At the next level, two subsequent blocks are concatenated and inserted into the HBF at level 1 and so on.

In the example shown, string "S0 S1 S2 S3" is blocked into blocks of size (s = 1) at the bottom of the hierarchy. Then "S0 S1" and "S2 S3" are inserted at level 1, and "S0 S1 S2 S3" at level 2. Thus, even if substrings have occurred at the appropriate offsets, going one level up in the hierarchy allows us to verify whether the substrings occurred together in the same or different emails.

Bloom filter with the offset of each element concatenated to it during insertion, like (content offset), improves the space utilization. For example, in order to store string "S0 S1 S2 S3" in the hierarchy, we need to insert the following strings into the Bloom filter {(S0 S1 S2 S3 0), (S0 S1 0), (S2 S3 1), (S0 0), (S1 1), (S2 2), (S3 3)}. Having a single Bloom filter allows us to maximize its space utilization as we can determine the optimal number of elements inserted into it a priori.

Furthermore, HBFs can also do limited pattern matching. Suppose we would like to verify if we have actually seen a string of the form “S0 S1 ∗ S3”. As in BBF, the string is broken down into three individual query strings {S0, S1, S3}. By trying all possible offsets at the bottom of the hierarchy we can verify the existence of strings {(S0 i), (S1 i + 1), (S3 i + 3) with false positive rate F P. Since ‘S0’ and ‘S1’ are subsequent in the query string we can improve the confidence of the results by verifying query string (S0 S1 i) at the level above. Now if we can make intelligent guesses for ‘*’ and when a match Sx is found, we can verify the match at different levels of the hierarchy. For example, we can verify the whole string “S0 S1 Sx S3” all the way to the top of the hierarchy consequently improving the confidence of the result at each level.

Saturation Point

A Bloom filter is a simple, space-efficient, randomized data structure for representing a set in order to support membership queries. It uses a set of k hash functions of range m and a bit vector of length m. Initially, the bit vector is set to 0. An element in the set is inserted into the Bloom filter by hashing the element using the k hash functions and setting the corresponding bits in the bit vector to 1. To test whether an element was inserted into the filter, we simply hash the element with the same hash functions and if all corresponding bits are set to 1 then the element is said to be present in the filter. The space efficiency of a Bloom filter is achieved at the cost of a small probability of false positives as defined by Equation, where n is the number of elements in the set

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

K – Number of hash functions.
m – Bit vector length.
n – Number of elements in the set.

The right hand side is minimized for K = ln 2 × m/n, in which case it becomes

$$\left(\frac{1}{2}\right)^k = (0.6185)^{m/n}.$$

From the below table, we have chosen k=4 and probability of false positive to be 0.003. We have chosen false positive rate as 0.003, because as the false positive rate is low, we will get accurate results. We have chosen bit vector length m=65536.

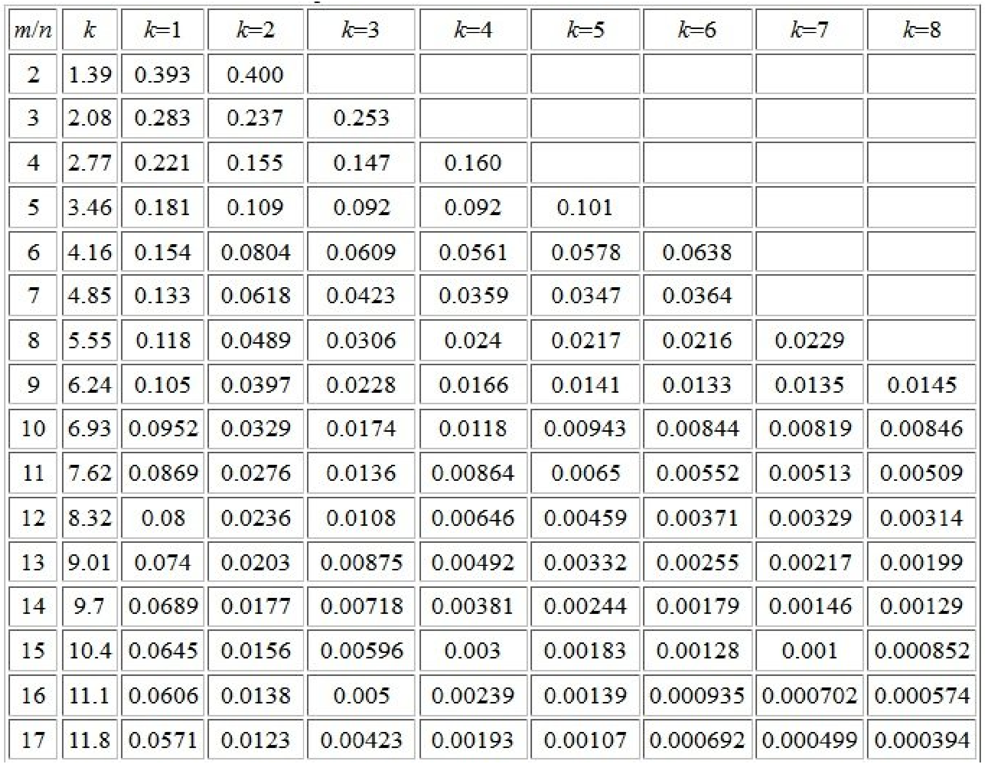
Known Parameters are
K = 4
Probability of false positive rate=0.003
m = 65536
n =?

Under K=4 and false positive rate of 0.003, m/n factor should be 15.we have chosen m to be 65536.

m/n = 15
65536/n =15
n=4369

So, Saturation Point is n=4369 (number of blocks that can be inserted into bloom filter).we chose block size to be 32 bytes.

The false positive ratios for common combinations of *m*/*n* and *k* are given below.



m/*n*	*k*	*k*=1	*k*=2	*k*=3	*k*=4	*k*=5	*k*=6	*k*=7	*k*=8
2	1.39	0.393	0.400						
3	2.08	0.283	0.237	0.253					
4	2.77	0.221	0.155	0.147	0.160				
5	3.46	0.181	0.109	0.092	0.092	0.101			
6	4.16	0.154	0.0804	0.0609	0.0561	0.0578	0.0638		
7	4.85	0.133	0.0618	0.0423	0.0359	0.0347	0.0364		
8	5.55	0.118	0.0489	0.0306	0.024	0.0217	0.0216	0.0229	
9	6.24	0.105	0.0397	0.0228	0.0166	0.0141	0.0133	0.0135	0.0145
10	6.93	0.0952	0.0329	0.0174	0.0118	0.00943	0.00844	0.00819	0.00846
11	7.62	0.0869	0.0276	0.0136	0.00864	0.0065	0.00552	0.00513	0.00509
12	8.32	0.08	0.0236	0.0108	0.00646	0.00459	0.00371	0.00329	0.00314
13	9.01	0.074	0.0203	0.00875	0.00492	0.00332	0.00255	0.00217	0.00199
14	9.7	0.0689	0.0177	0.00718	0.00381	0.00244	0.00179	0.00146	0.00129
15	10.4	0.0645	0.0156	0.00596	0.003	0.00183	0.00128	0.001	0.000852
16	11.1	0.0606	0.0138	0.005	0.00239	0.00139	0.000935	0.000702	0.000574
17	11.8	0.0571	0.0123	0.00423	0.00193	0.00107	0.000692	0.000499	0.000394

Querying

The hierarchical nature of the HBF resolves collisions automatically. Suppose we would like to verify if we have actually seen a string of the form "S0S1S2 S3". As in BBF, the string is broken down into four individual query strings {S0, S1, S2, S3}. By trying all possible offsets at the bottom of the hierarchy we can verify the existence of strings {(S0 || i), (S1 || i + 1), (S2 || i + 2), (S3 || i + 3) with false positive rate FP. Since 'S0' and 'S1' are subsequent in the query string we can improve the confidence of the results by verifying query string (S0S1||i) at the level above. In this way we can verify the matches at different levels of the hierarchy. For example, we can verify the whole string "S0S1S2S3" all the way to the top of the hierarchy consequently improving the confidence of the result at each level.

References

1. http://en.wikipedia.org/wiki/Sendmail
2. http://en.wikipedia.org/wiki/Milter
3. http://en.wikipedia.org/wiki/Bloom_filter
4. https://www.milter.org/developers
5. https://www.milter.org/developers/api/index
6. http://milter-manager.sourceforge.net/reference/introduction.html
7. Payload Attribution via Hierarchical Bloom Filters
8. Gimp Toolkit