ARTICLE IN PRESS

Computer Networks xxx (2010) xxx–xxx

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

Improving retouched Bloom filter for trading off selected false positives against false negatives

Benoit Donnet [a,*], Bruno Baynat [b,**], Timur Friedman [b,***]

[a] Université catholique de Louvain, Louvain-la-Neuve, Belgium
[b] UPMC Sorbonne Universités and CNRS, Paris, France

ARTICLE INFO

Article history:
Received 8 September 2009
Received in revised form 4 June 2010
Accepted 8 July 2010
Available online xxxx

Responsible Editor: C. Westphal

Keywords:
Bloom filters
False positives
False negatives
Bit clearing
Measurement
Traceroute

ABSTRACT

Where distributed agents must share voluminous set membership information, Bloom filters provide a compact, though lossy, way for them to do so. Numerous recent networking papers have examined the trade-offs between the bandwidth consumed by the transmission of Bloom filters, and the error rate, which takes the form of false positives. This paper is about the retouched Bloom filter (RBF). An RBF is an extension that makes the Bloom filter more flexible by permitting the removal of false positives, at the expense of introducing false negatives, and that allows a controlled trade-off between the two. We analytically show that creating RBFs through a random process decreases the false positive rate in the same proportion as the false negative rate that is generated. We further provide some simple heuristics that decrease the false positive rate more than the corresponding increase in the false negative rate, when creating RBFs. These heuristics are more effective than the ones we have presented in prior work. We further demonstrate the advantages of an RBF over a Bloom filter in a distributed network topology measurement application. We finally discuss several networking applications that could benefit from RBFs instead of standard Bloom filters.

1. Introduction

Introduced in 1970 [1], it is just in the past decade that the *Bloom filter* has attracted attention from the networking research community [2]. A Bloom filter compactly encodes set information into a bit vector that can then be queried regarding set membership. A vector of all zeroes represents the empty set. To record a key as being in the set, hash it to obtain an index into the vector and set the bit at that position to one. You may use multiple hash functions, in which case you set several bits to one. To query if a key is in the set, check if all hash positions are set to one. Though the filter will occasionally return a false positive, erroneously claiming that a key belongs to the set, it will never return a false negative, erroneously claiming that a key does not belong. You may set the vector size and number of hash functions in light of the anticipated set size, to aim for a particular trade-off between the size of the bit vector and the false positive rate.

A prime appeal of the Bloom filter to networking researchers comes from the bandwidth efficiencies that it offers for the transmission of set membership information between networked hosts [3]. We ourselves have proposed their use for large-scale route tracing infrastructures [4]. Continuously running production systems of this sort include Archipelago [5], DIMES [6], RIPE TTM [7], iPlane [8], Gulliver [9], Ono [10], and TDMI [11]. They have in common the placement of agents across the Internet so as to obtain measurements from a variety of vantage points. If these agents are to coordinate their efforts, as Archipelago agents will do for their *scamper* measurements [12], then they must communicate between each other, either directly or

* Corresponding author. Tel.: +32 10 478718; fax: +32 10 450345.
** Corresponding author.
*** Corresponding author.
E-mail addresses: benoit.donnet@uclouvain.be (B. Donnet), bruno.baynat@lip6.fr (B. Baynat), timur.friedman@lip6.fr (T. Friedman).

1389-1286/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.comnet.2010.07.003

via a central server. Such communication is potentially bandwidth-hungry, which is a problem if the agents are expected to adhere to tight bandwidth constraints, as they are for Ono [13, Section 5.3].

This paper describes a way for distributed route tracing agents to coordinate to reduce their impact on end-hosts. For each trace that an agent makes, it records the IP address that it encounters one hop before the end host. We call this the *penultimate node*. The agent encodes the set of penultimate nodes in a Bloom filter, which it sends to other agents. Another agent consults the filter while tracing: if it encounters an address already seen by the first agent, it stops tracing that route and goes onto another one.

Using Bloom filters in this way allows a trade-off between filter size and coordination failures. A smaller filter saves on the bandwidth required for coordination. A larger filter reduces on average the rate of false positives, which stop route tracing before the penultimate node is reached and thereby deprive the system of potentially useful information. Without the Bloom filter, there would be no such flexibility. Replacing the filter with an explicit list of penultimate nodes would limit the infrastructure operator to one extreme end of this trade-off: high coordination bandwidth in exchange for no traces that stop too early. This, despite the fact that some low level of coordination failures might perhaps be tolerable.

Researchers have proposed the Bloom filter for so many networked applications precisely to allow them to enjoy this sort of flexible trade-off. However we can do even better, as this trade-off is more simplistic and limiting than it needs to be. Simplistic, because the false positive rate expresses an average based on an idealized query distribution in which all keys are equiprobable, whereas actual system performance depends on that distribution and the identities of those keys that cause the false positives. Limiting, because the system might tolerate false negatives, but the Bloom filter does not allow us to introduce false negatives into the trade-off.

This paper describes the *retouched Bloom filter* (RBF), a modification to the standard Bloom filter that allows us to remove selected false positives at the cost of introducing random false negatives. We create an RBF from a Bloom filter, as Section 3 describes, by selectively changing individual bits from 1 to 0, while the size of the filter and the query mechanism remain unchanged. As Section 3.1 shows analytically, if we create an RBF through a purely random process then we decrease the false positive rate, on average, in the same proportion as the false negative rate that we generate. Simple heuristic algorithms that we present in Section 3.1 do better than the random process and lower the false positive rate by a greater degree, on average, than the corresponding increase in the false negative rate. In addition, Section 4 provides mechanisms that allow us to selectively remove the most troublesome false positives, further improving performance when we take the query distribution into account.

The RBF algorithms require space that is at most a small constant multiple of the Bloom filter's vector size. Compared to the creation of a standard Bloom filter, the RBF algorithms also incur additional processing costs related to key removal. These costs are a constant multiple of a number of RBF parameters, such as the number of hash functions and the number of false positives to remove. The additional processing and storage requirements that are incurred when switching from Bloom filters to RBFs are restricted entirely to the locations at which the RBFs are created. There is strictly no addition to the critical resource in our networked scenario, which is the bandwidth consumed by communication between measurement points. At the receiver of the RBF, queries take place using exactly the same mechanism as for the Bloom filter, incurring no additional space or time complexity.

Compared to our previous work [4], which introduced the RBF, the algorithms in this paper are more effective. By more carefully tracking the quantities of false negatives generated and false positives removed at each step of an algorithm, we achieve a greater decrease, on average, in the false positive rate for a given increase in the false negative rate. Based on simulations, we demonstrate that our new improved algorithms perform between 10% and 20% better on average than the simple algorithms we previously proposed. The case study in Section 5 has been rerun using these improved algorithms.

The work that we present here is the first that subjects false negatives in a Bloom filter variant to either analytic or simulation studies. In particular, our work is the first to explicitly study the trade-off between false positives and false negatives and it is the first to consider the efficiency of the means employed for such a trade-off. Section 6 describes the extensive related work on Bloom filters [3,14–20]. Some of these extensions, such as the *anti-Bloom filter* [14] and the *generalized Bloom filter* [15], target the suppression of false positives, or the removal of bits in the vector in general. Some of them (such as the *variable Bloom filter* [19]) even provide a trade-off between the false positive rate and the false negative rate. Nevertheless, as we show, these variants differ significantly from the standard Bloom filter in that they either increase the memory cost (i.e., increase the size of the filter) or they modify the filter behavior when performing membership queries.

The remainder of this paper is organized as follows: Section 2 presents the standard Bloom filter, using notation introduced by Broder and Mitzenmacher [2]; Section 3 presents the RBF, and shows analytically that the reduction in the false positive rate is equal, on average, to the increase in the false negative rate even as random 1s in a Bloom filter are reset to 0s; Section 4 presents improved methods for selectively clearing 1s that are associated with the most troublesome false positives, and shows through simulations that they reduce the false positive rate by more, on average, than they increase the false negative rate; Section 5 describes the use of RBFs in a network measurement application; Section 6 discusses several Bloom filter variants, compares RBFs to them and discusses other networking usages of RBFs; finally, Section 7 summarizes the conclusions and future directions for this work.

2. Bloom filters

A *Bloom filter* [1] is a vector v of m bits that codes the membership of a subset $A = \{a_1, a_2, \ldots, a_n\}$ of n elements of

Please cite this article in press as: B. Donnet et al., Improving retouched Bloom filter for trading off selected false positives against false negatives, Comput. Netw. (2010), doi:10.1016/j.comnet.2010.07.003

an universe U consisting of N elements. In most papers, the size of the universe is not specified. However, Bloom filters are only useful if the size of U is much larger than the size of A.

We initialize this vector v to 0, and then operate on it using a set $H = \{h_1, h_2, \ldots, h_k\}$ of k independent hash functions, each with range $\{1, \ldots, m\}$. For each element $a \in A$, we set the bits at positions $h_1(a), h_2(a), \ldots, h_k(a)$ in v to 1. Note that a particular bit can be set to 1 several times, as illustrated in Fig. 1.

To check if an element b of the universe U belongs to the set A, all one has to do is check that the k bits at positions $h_1(b), h_2(b), \ldots, h_k(b)$ are all set to 1. If *at least* one bit is set to 0, we are sure that b does not belong to A. If *all* bits are set to 1, b possibly belongs to A. There is always a probability that b does not belong to A. In other words, there is a risk of *false positives*. Let us denote by F_P the set of false positives, i.e., the elements that do not belong to A (and thus that belong to $U - A$) and for which the Bloom filter gives a positive answer. The sets U, A and F_P are illustrated in Fig. 2. (B is a subset of F_P that will be introduced below.) In Fig. 2, F_P is a circle surrounding A. (Note that F_P is not a superset of A. It has been colored distinctly to indicate that it is disjoint from A.) Fig. 3 gives an example of a false positive.

We define the *false positive proportion* f_P as the ratio of the number of elements in $U - A$ that give a positive answer, to the total number of elements in $U - A$:

$$f_P = \frac{|F_P|}{|U - A|}. \tag{1}$$

We can alternatively define the *false positive rate*, as the probability that for a given element that does not belong to the set A, the Bloom filter erroneously claims that the element is in the set. Note that if this probability exists, it has the same value as the false positive proportion f_P. As a consequence, we will use the same notation for both parameters and also denote by f_P the false positive rate. In order to calculate the false positive rate, most papers assume that all hash functions map each item in the universe to a random number uniform over the range $\{1, \ldots, m\}$. As a consequence, the probability that a specific bit is set to 1 after the application of one hash function to one element of A is $\frac{1}{m}$ and the probability that this specific bit is left to 0 is $1 - \frac{1}{m}$. After all elements of A are coded in the Bloom filter, the probability that a specific bit is always equal to 0 is

$$p_0 = \left(1 - \frac{1}{m}\right)^{kn}. \tag{2}$$

As m becomes large, $\frac{1}{m}$ is close to zero and p_0 can be approximated by

$$p_0 \approx e^{-\frac{kn}{m}}. \tag{3}$$

The probability that a specific bit is set to 1 can thus be expressed as

$$p_1 = 1 - p_0. \tag{4}$$

The false positive rate can then be estimated by the probability that each of the k array positions computed by the hash functions is 1. f_P is then given by

$$f_P = p_1^k = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-\frac{kn}{m}}\right)^k. \tag{5}$$

The false positive rate f_P is thus a function of three parameters: n the size of subset A, m the size of the filter, and k the number of hash functions. Fig. 4 illustrates the variation of f_P with respect to the three parameters individually (when the two others are held constant). Obviously, and as can been seen in these graphs, f_P is a decreasing function of m and an increasing function of n. Now, when k varies (with n and m constant), f_P first decreases, reaches a minimum and then increases. There are two countervailing factors: using more hash functions gives us more chances to find a *0* bit for an element that is not a member of A, but using fewer hash functions increases the fraction of *0* bits in the array. As stated, e.g., by Broder and Mitzenmacher [2], f_P is minimized when

$$k = \frac{m \ln 2}{n}, \tag{6}$$

for fixed m and n.

Thus the minimum possible false positive rate for given values of m and n is given by Eq. (7). In practice, of course, k must be an integer. As a consequence, Eq. (6) has to be

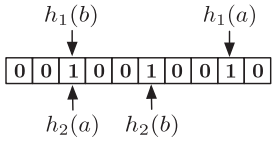

Fig. 1. A Bloom filter with two hash functions.

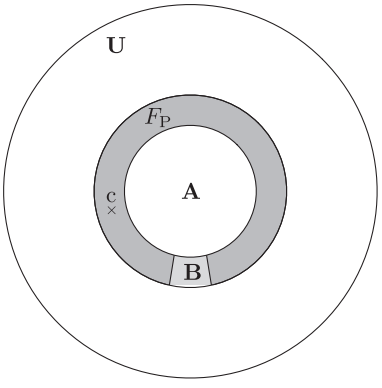

Fig. 2. The false positives set.

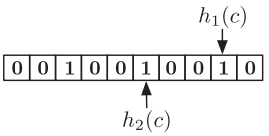

Fig. 3. An example of false positive.

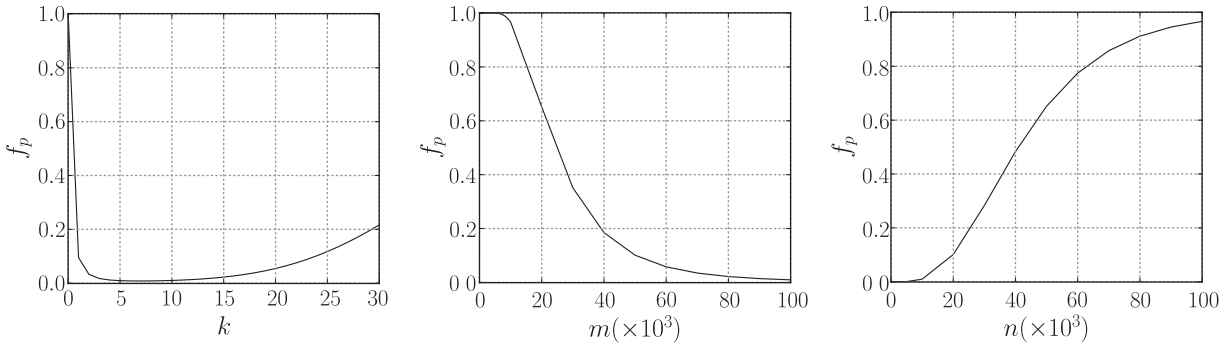

Fig. 4. f_P as a function of k, m and n.

rounded to the nearest integer and the resulting false positive rate will be a bit higher than the optimal value given in Eq. (7)

$$\hat{f}_P = \left(\frac{1}{2}\right)^{\frac{m \ln 2}{n}} \approx (0.6185)^{\frac{m}{n}}. \quad (7)$$

Finally, it is important to emphasize that the absolute number of false positives is relative to the size of $U - A$ (and not directly to the size of A). This result seems surprising as the expression of f_P depends on n, the size of A, and does not depend on N, the size of U. If we double the size of $U - A$ (and keep the size of A constant) we also double the absolute number of false positives (and obviously the false positive rate is unchanged).

3. Retouched Bloom filters

As shown in Section 2, there is a trade-off between the size of the Bloom filter and the probability of a false positive. For a given n, even by optimally choosing the number of hash functions, the only way to reduce the false positive rate in standard Bloom filters is to increase the size m of the bit vector. Unfortunately, although this implies a gain in terms of a reduced false positive rate, it also implies a loss in terms of increased memory usage. Bandwidth usage becomes a constraint that must be minimized when Bloom filters are transmitted in the network.

This paper describes an extension to the Bloom filter, referred to as the *retouched Bloom filter* (RBF).[1] The RBF makes standard Bloom filters more flexible by allowing false positives to be traded off against false negatives, which do not arise at all in the standard case. The idea behind the RBF is to remove a certain number of false positives by resetting bits in vector v. We call this process the *bit clearing process*. Resetting a given bit to 0 not only has the effect of removing a certain number of false positives, but also generates false negatives. Indeed, any element $a \in A$ such that (at least) one of the k bits at positions $h_1(a), h_2(a), \ldots, h_k(a)$ has been reset to 0, now triggers a negative answer, so becoming a false negative.

[1] A Java implementation of retouched Bloom filters and other interesting Bloom filter variations is freely available at http://gforge.info.ucl.ac.be/projects/filters/, under a BSD-like license.

To summarize, the bit clearing process has the effects of decreasing the number of false positives and of generating a number of false negatives. Let us use the labels F'_P and F'_N to describe the sets of false positives and false negatives after the bit clearing process. The sets F'_P and F'_N are illustrated in Fig. 5.

After the bit clearing process, the false positive and false negative proportions are given by

$$f'_P = \frac{|F'_P|}{|U - A|}, \quad (8)$$

$$f'_N = \frac{|F'_N|}{|A|}. \quad (9)$$

Obviously, the false positive proportion has decreased (as F'_P is smaller than F_P) and the false negative proportion has increased (as it was zero before the clearing). We can measure the benefit of the bit clearing process by introducing Δf_P, the proportion of false positives removed by the bit clearing process, and Δf_N, the proportion of elements of A that becomes false negatives after the bit clearing process:

$$\Delta f_P = \frac{|F_P| - |F'_P|}{|F_P|} = \frac{f_P - f'_P}{f_P}, \quad (10)$$

$$\Delta f_N = \frac{|F'_N|}{|A|} = f'_N. \quad (11)$$

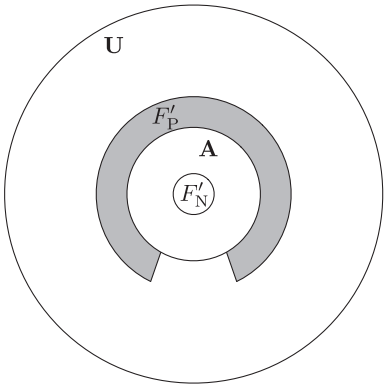

Fig. 5. False positive and false negative sets after the selective clearing process.

We, finally, define χ as the ratio between the proportion of false positives removed and the proportion of false negatives generated:

$$\chi = \frac{\Delta f_P}{\Delta f_N}. \quad (12)$$

χ is the main metric we use in this paper to evaluate the RBF. If χ is greater than 1, it means that the proportion of false positives removed is higher than the proportion of false negatives generated.

3.1. Randomized bit clearing

This section analytically examines the effect of randomly resetting bits in the Bloom filter, whether these bits correspond to false positives or not. We call this process the *randomized bit clearing process*. Section 4 will discuss more sophisticated approaches to selecting the bits that should be cleared. However, performing random clearing in the Bloom filter enables us to derive analytical results concerning the consequences of the clearing process. In addition to providing a formal derivation of the benefit of retouched Bloom filters, it also gives a lower bound on the performance of any smarter selective clearing approach (such as those developed in Section 4).

We again assume that all hash functions map each element of the universe U to a random number uniform over the range $\{1,\ldots,m\}$. Once the n elements of A have been coded in the Bloom filter, there is a probability p_0 for a given bit in v to be *0* and a probability p_1 for it to be *1*. As a consequence, there is an average number of $p_1 m$ bits set to *1* in v. Let us study the effect of resetting to *0* a randomly chosen bit in v. Each bit set to 1 in v has a probability $\frac{1}{p_1 m}$ of being reset and a probability $1 - \frac{1}{p_1 m}$ of being left at *1*.

The first consequence of resetting a bit to *0* is to remove a certain number of false positives. If we consider a given false positive $x \in F_P$, after the reset it will not result in a positive test any more if the bit that has been reset belongs to one of the k positions $h_1(x), h_2(x), \ldots, h_k(x)$. Conversely, if none of the k positions have been reset, x remains a false positive. The probability of this latter event is

$$r_1 = \left(1 - \frac{1}{p_1 m}\right)^k. \quad (13)$$

As a consequence, after the reset of one bit in v, the false positive rate decreases from f_P (given by Eq. (5)) to $f'_P = f_P r_1$. The proportion of false positives that have been eliminated by the resetting of a randomly chosen bit in v is thus equal to $1 - r_1$:

$$\Delta f_P = 1 - r_1. \quad (14)$$

The second consequence of resetting a bit to *0* is the generation of a certain number of false negatives. If we consider a given element $a \in A$, after the reset it will result in a negative test if the bit that has been reset in v belongs to one of the k positions $h_1(a), h_2(a), \ldots, h_k(a)$. Conversely, if none of the k positions have been reset, the test on a remains positive. Obviously, the probability that a given element in A becomes a false negative is given by $1 - r_1$ (the same reasoning holds):

$$\Delta f_N = 1 - r_1. \quad (15)$$

We have demonstrated that resetting one bit to *0* in v has the effect of eliminating the same proportion of false positives as the proportion of false negatives generated. As a result, $\chi = 1$. It is however important to note that the proportion of false positives that are eliminated is relative to the size of the set of false positives (which in turns is relative to the size of $U - A$, thanks to Eq. (1)) whereas the proportion of false negatives generated is relative to the size of A. As we assume that $U - A$ is much bigger than A (actually if $|F_P| > |A|$), resetting a bit to *0* in v can eliminate many more false positives than the number of false negatives generated.

It is easy to extend the demonstration to the reset of s bits and see that it eliminates a proportion $1 - r_s$ of false positives and generates the same proportion of false negatives, where r_s is given by

$$r_s = \left(1 - \frac{s}{p_1 m}\right)^k. \quad (16)$$

As a consequence, any random clearing of bits in the Bloom vector v has the effect of maintaining the ratio χ equal to *1*.

4. Selective clearing

Section 3 introduced the idea of randomized bit clearing and analytically studied the effect of randomly resetting s bits of v, whether these bits correspond to false positives or not. We showed that it has the effect of maintaining the ratio χ equal to 1. This section refines the idea of randomized bit clearing by focusing on bits corresponding to elements that trigger false positives. We call this refined process *selective clearing*.

4.1. Objectives

Section 2 explained that Bloom filters might trigger false positives (i.e., a key is erroneously claimed as belonging to the filter), forming the set F_P. In practice, it is likely that not all false positives will be encountered. In addition, some false positives, which we call *troublesome keys*, might be encountered more frequently than others. We record troublesome keys in a set called B, with $B \subseteq F_P$ (see Fig. 2).

The purpose of selective clearing is to remove from the filter elements belonging to B. To this end, we proposed in our previous work four different algorithms: *Random Selection*, *Minimum FN Selection*, *Maximum FP Selection*, and *Ratio Selection* [4]. These each employ heuristics aimed at being more effective than randomized bit clearing.

Random Selection randomly selects, for each troublesome key to remove, a bit amongst the k available to reset. Random Selection differs from random clearing (see Section 3) by focusing on a set of troublesome keys to remove, B, and not by randomly resetting any bit in v, whether it corresponds to a false positive or not. Minimum FN Selection aims at minimizing the false negatives generated by each selective clearing. Maximum FP Selection aims at maximizing the quantity of false positives to remove. Finally, Ratio Selection combines Minimum FN Selection

Please cite this article in press as: B. Donnet et al., Improving retouched Bloom filter for trading off selected false positives against false negatives, Comput. Netw. (2010), doi:10.1016/j.comnet.2010.07.003

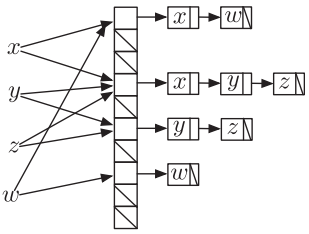

Fig. 6. Example of an ElementList vector.

and Maximum FP Selection into a single algorithm. Ratio Selection aims to minimize the false negatives generated while maximizing the false positives removed.

Except for Random Selection, all selective clearing algorithms make use of one or two counting vectors during the selective clearing process. To keep the algorithms simple, these counting vectors are not updated with each iteration of each algorithm. The cost of this simplicity comes in terms of an over-estimation, for the heuristic, in assessing the number of false negatives that it introduces in any given iteration. This over-estimation grows as the algorithm progresses.

Section 4.2 describes three improved heuristic selective clearing algorithms that keep up-to-date the quantity of false negatives generated and false positives removed at each step of the algorithms. This is done using a data structure that we call such an *ElementList vector*, illustrated in Fig. 6. It is somewhat similar to the fast hash tables developed by Song et al. [16]. The vector has the same length as the bit vector. It contains thus m cells. Each cell is a pointer to a list of elements recorded in that position in the bit vector. These elements, depending on the selective clearing algorithm, can belong to A or B.

The additional cost of our improved algorithms come from the spatial complexity of each algorithm. Compared to the simple algorithms, our improved selective clearing algorithms need more space to store the associated ElementList vectors. Recall also that any additional processing and storage related to the creation of RBFs from Bloom filters are restricted to the entity creating the RBFs. Once an RBF is created, we only work with the bit vector, as standard Bloom filters do.

4.2. Improved algorithms

All algorithms discussed in this section assume that the function MEMBERSHIPTEST is defined. It takes two arguments: the key to be tested and the bit vector. This function returns *true* if the element is recorded in the bit vector (i.e., all the k positions corresponding to the hash functions are set to 1). It returns *false* otherwise.

Algorithm 1 Improved Minimum FN Selection

Require: v, the bit vector and v_A, the ElementList vector
Ensure: v and v_A updated, if needed

```
1:  procedure MinimumFNSelection (v,A,B)
2:    v_A ← create (A)
3:    for All b_i ∈ B do
4:      if membershiptest (b_i, v) then
5:        index ← minindex (b_i, v_A)
6:        bitclearing (v_A, index)
7:        v[index] ← 0
8:      end if
9:    end For
10: end procedure
11:
12: procedure create (A)
13:   ElementListVector v
14: for all a_i ∈ A do
15:     for j = 1 to k do
16:       v[h_j(a_i)].add (a_i)
17:     end for
18:   end for
19:   return v
20: end procedure
21:
22: procedure bitclearing (v, index)
23:   ElementList el = v.get (index)
24:   for All x_i ∈ el do
25:     remove (x_i, v)
26:   end for
27: end procedure
```

The first algorithm, *Improved Minimum FN Selection*, is formally defined in Algorithm 1 and aims, for any troublesome key to remove, at selecting a bit amongst the k available the one that will generate the minimum amount of false negatives. It makes use of an ElementList vector, v_A, each cell containing the list of elements belonging to A that are recorded in the corresponding cell of v, the bit vector. For each troublesome key to remove that was not previously cleared, we choose amongst the k bit positions the one that we estimate will generate the minimum number of false negatives. This minimum is given by the MININDEX procedure using the ElementList vector v_A and running through the list of elements stored in indicated positions. When the minimum index has been returned by MININDEX, the Improved Minimum FN Selection algorithm call the BITCLEARING procedure that will remove from v_A all the elements recorded in this minimum index.

Table 1
Individual performance of selective clearing when removing 5% ($\beta = 0.05$) of false positives.

Algorithm	$\lvert B\rvert$	$\lvert B'\rvert$	$\lvert B\rvert + \lvert B'\rvert$	$\lvert A'\rvert$
Random	932 ± 9	1934 ± 37	2826 ± 46	1070 ± 10
Improved Minimum FN	939 ± 5	1919 ± 36	2858 ± 40	788 ± 8
Improved Maximum FP	943 ± 6	3275 ± 38	4218 ± 43	1043 ± 11
Improved ratio	937 ± 6	3082 ± 45	4020 ± 51	843 ± 6

Please cite this article in press as: B. Donnet et al., Improving retouched Bloom filter for trading off selected false positives against false negatives, Comput. Netw. (2010), doi:10.1016/j.comnet.2010.07.003

The algorithmic complexity of the Improved Minimum FN Selection is $O(k \times (|A| + |B|))$. Indeed, the algorithm starts by creating v_A (procedure in $O(k \times |A|)$) and, next, for each troublesome key belonging to B, it calls MEMBERSHIPTEST. In the case of a positive return MEMBERSHIPTEST, it looks for the minimum index and, then, performs the bit clearing. Note that the algorithmic complexity of the cumulated calls of BITCLEARING cannot be worst than the algorithmic complexity of CREATECV (clearing the ElementList vector is not harder, in a complexity sense, than creating it). Based on this, we see that the algorithmic complexity of Improved Minimum FN Selection is $O(\max(k \times |A|, 2k \times |B|))$, which leads to $O(k \times (|A| + |B|))$.

Algorithm 2 Improved Maximum FP Selection

Require: v, the bit vector and v_B, the ElementList vector
Ensure: v and v_B updated, if needed
1: **procedure** MAXIMUMFP (v,A,B)
2: $v_B \leftarrow$ CREATE (B)
3: **for all** $b_i \in B$ **do**
4: **if** MEMBERSHIPTEST (b_i, v) **then**
5: index $\leftarrow$ MAXINDEX (b_i)
6: BITCLEARING (v_B, index)
7: v[index] $\leftarrow$ 0
8: **end if**
9: **end for**
10: **end procedure**

The second algorithm, *Improved Maximum FP Selection*, is given in Algorithm 2. For any troublesome key to remove, it selects a bit amongst the k available that will remove the largest number of false positives. This is done using the MAXINDEX function along with the ElementList vector, v_B, that records the troublesome keys registered in the bit vector v. When the maximum index is found by the MAXINDEX procedure, the BITCLEARING procedure is called in order to maintain v_B up-to-date.

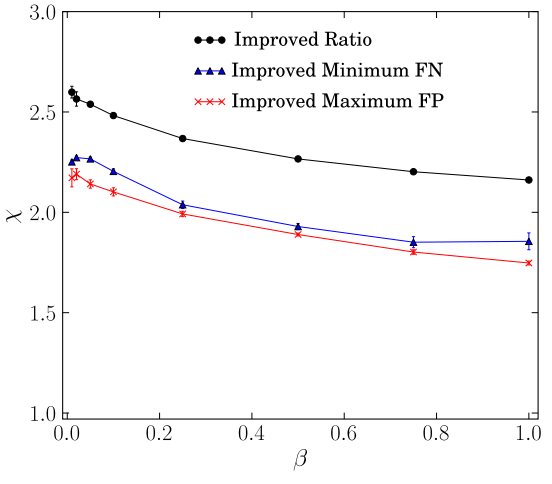

Fig. 7. Performances of improved selective clearing algorithms.

The algorithmic complexity of the Improved Maximum FP Selection is $O(k \times |B|)$. The reasoning for the running time of Improved Maximum FP Selection is identical to Improved Minimum FP. The only difference stands in the fact that the CREATE procedure concerns the set B.

Algorithm 3 Improved Ratio Selection

Require: v, the bit vector, v_B and v_A, the ElementList vectors and r, the ratio vector
Ensure: v, v_A, v_B and r updated, if needed
1: **procedure** RATIO (v,A,B)
2: $v_A \leftarrow$ CREATE (A)
3: $v_B \leftarrow$ CREATE (B)
4: COMPUTERATIO ()
5: **for all** $b_i \in B$ **do**
6: **if** MEMBERSHIPTEST (b_i, v) **then**
7: index $\leftarrow$ MINRATIO (b_i)
8: BITCLEARING (v_A, index)
9: BITCLEARING (v_B, index)
10: v[index] $\leftarrow$ 0
11: r[index] $\leftarrow$ 0
12: COMPUTERATIO ()
13: **end if**
14: **end for**
15: **end procedure**
16:
17: **procedure** COMPUTERATIO
18: **for** i = 1 to m **do**
19: **if** v[i] $\wedge$ $v_B[i]$.*size* () > 0 **then**
20: r[i] $\leftarrow \frac{v_A[i].size()}{v_B[i].size()}$
21: **end if**
22: **end for**
23: **end procedure**

Our last algorithm, called *Improved Ratio Selection* (see Algorithm 3), combines Improved Minimum FN Selection and Maximum FP Selection into a single algorithm. Improved Ratio Selection provides an approach in which we try to minimize the false negatives generated while maximizing the false positives removed. Improved Ratio Selection therefore takes into account the risk of collision between hashed keys of elements belonging to A and hashed keys of elements belonging to B. Further, the ratio vector, r, containing the ratio of the number of elements recorded in a given cell of v_A to the number of elements recorded in a given cell of v_B is also maintained up-to-date. This is achieved by calling the RATIO procedure each time a false positive is removed from the bit vector.

The algorithmic complexity of Improved Ratio Selection is $O(k \times (|A| + |B|) + m)$. The reasoning for the running time of Improved Maximum FP Selection is identical to Improved Minimum FP, the factor m coming from the COMPUTERATIO procedure.

4.3. Simulation analysis

4.3.1. Methodology

We conducted an experiment with an universe U of 2,000,000 elements (N = 2,000,000). These elements, for

Please cite this article in press as: B. Donnet et al., Improving retouched Bloom filter for trading off selected false positives against false negatives, Comput. Netw. (2010), doi:10.1016/j.comnet.2010.07.003

the sake of simplicity, were integers belonging to the range [0; 1,999,9999]. The subset A that we wanted to summarize in the Bloom filter contains 10,000 different elements (n = 10,000) randomly chosen from the universe U.

The bit vector v we used for simulations is 100,000 bits long (m = 100,000), ten times larger than $|A|$. The RBF used five different and independent hash functions (k = 5). Hashing was emulated with random numbers. We simulated randomness with the Mersenne Twister MT19937 pseudo-random number generator [21]. Using five hash functions and a bit vector ten times bigger than n is advised by Fan et al. [3]. This permits a good trade-off between membership query accuracy, i.e., a low false positive rate of 0.0094 when estimated with Eq. (5), memory usage and computation time.

For our experiment, we defined the ratio of troublesome keys compared to the entire set of false positives as

$$\beta = \frac{|B|}{|F_P|}. \quad (17)$$

We considered the following values of β: 1%, 2%, 5%, 10%, 25%, 50%, 75% and 100%. When β = 100%, it means that $B = F_P$ and we want to remove all the false positives.

Each data point in the plots represents the mean value over fifteen runs of the experiment, each run using a new A, F_P, B, and RBF. We determined 95% confidence intervals for the mean based on the Student t distribution.

We performed the experiment as follows: we first created the universe U and randomly affected 10,000 of its elements to A. We next built F_P by applying the following scheme. Rather than using Eq. (5) to compute the false positive rate and then creating F_P by randomly affecting positions in v for the false positive elements, we preferred to experimentally compute the false positives. We queried the RBF with a membership test for each element belonging to $U - A$. False positives were the elements that belong to the Bloom filter but not to A. We kept track of them in a set called F_P. This process seemed to us more realistic because we evaluated the real quantity of false positive elements in our data set. B was then constructed by randomly selecting a certain quantity of elements in F_P, the quantity corresponding to the desired cardinality of B. We next removed all troublesome keys from B by using one of the improved selective clearing algorithms, as explained in Section 4.2. We then built F'_N, the false negative set, by testing all elements in A and adding to F'_N all elements that no longer belong to A. We also determined F'_P, the false positive set after removing the set of troublesome keys B.

4.3.2. Results

We first evaluate each algorithm individually. Next we will compare our improved algorithms to the previous versions [4].

Our individual evaluation is based on the four following metrics:

- $|B|$, the number of troublesome keys to remove.
- $|B'|$, the number of additional false positive keys removed. This is thus the side effect of performing selective clearing.
- $|B + B'|$, summarizes the two previous metrics, giving the total number of false positives removed.
- $|A'|$, the quantity of keys that become false negatives after selective clearing.

Table 1 shows the performance of each selective clearing algorithm considering the four metrics when we remove 5% (i.e., β = 0.05) of the false positives from the filter. Results given are an average over fifteen runs. The confidence interval around the mean is also provided.

As indicated by the column labeled "$|B'|$", the side effect of removing false positives strongly differs between the algorithms. Indeed, Improved Maximum FP and Improved Ratio remove more than 3,000 additional false positives compared to a lower 2,000 for other algorithms. This behavior is expected by definition of Improved Maximum FP as it tries to maximize the amount of false positives removed at each step of the algorithm.

When considering the amount of false negatives generated by removing false positives, we note that Improved Minimum FN and Improved Ratio act better. This is expected by definition of Improved Minimum FN (i.e., minimizing the amount of false negatives generated at each step).

To summarize results provided in Table 1, one can say that Improved Ratio performs best by trying to combine in a single algorithm Improved Minimum FN and Improved Maximum FP. Indeed, the side effect of removing additional false positives at each step is high, while Improved Ratio keeps the amount of false negatives generated reasonably low.

Fig. 7 deeper evaluates the performance of our improved selective clearing algorithms. It plots χ, defined by Eq. (12), in function of β, defined in Eq. (17). The confidence intervals are plotted but they are generally too tight to be visible.

In the fashion of simple selective clearing algorithms presented in our previous work [4], whatever the algorithm considered, the χ ratio is above 1, meaning that the advantages of removing false positives overcome the drawbacks of generating false negatives, if these errors are considered equally grave. Thus, as expected, performing selective clearing provides better results than randomized bit clearing. Another general tendency is that, whatever the algorithm, the performance slightly decreases when β increases but still remains largely above one.

If we look now at each algorithm in particular, we notice that Improved Ratio does best. This is easily understandable as, by definition, it combines the advantages of both Improved Minimum FN and Improved Maximum FP, i.e., it tries to minimizes the false negatives generated while, at the same time, maximizing the false positives removed.

Fig. 8 proposes a comparison between simple and improved selective clearing algorithms. The vertical axis presents the relative difference between the χ for simple and improved algorithms. It is given by:

$$\text{Relative difference} = \frac{\chi_{\text{improved}} - \chi_{\text{simple}}}{\chi_{\text{simple}}}. \quad (18)$$

Please cite this article in press as: B. Donnet et al., Improving retouched Bloom filter for trading off selected false positives against false negatives, Comput. Netw. (2010), doi:10.1016/j.comnet.2010.07.003

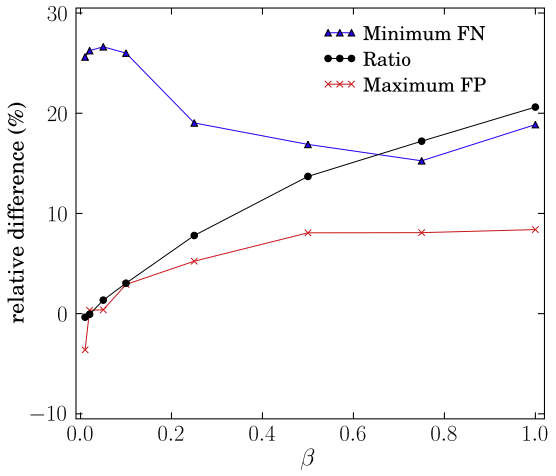

Fig. 8. Relative difference between simple and improved selective clearing.

From Fig. 8, we see that, with respect to our performance criteria, our improved selective algorithms do between 10% and 20% better (in general) than simple algorithms previously proposed. We believe that the cost associated to this performance increase, i.e., a slight increase in memory usage when creating the filter, is negligible compared to the benefits. Further, once the RBF has been created, the standard behavior of a Bloom filter is maintained. The complexity takes place only during the filter construction process.

Finally, given the results discussed in this section, we advice the use of the Improved Ratio Algorithm as it provides the best performance.

5. Case study

5.1. Tracing paths with a red stop set

Retouched Bloom filters can be applied across a wide range of applications that would otherwise use Bloom filters (see Section 6.2). For RBFs to be suitable for an application, two criteria must be satisfied. First, the application must be capable of identifying instances of false positives. Second, the application must accept the generation of false negatives, and in particular, the marginal benefit of removing the false positives must exceed the marginal cost of introducing the false negatives.

This section describes the application that motivated our introduction of RBFs: a network measurement system that traces routes, and must communicate information concerning the IP addresses at which to stop tracing. Section 5.2 evaluates the impact of using RBFs in this application.

Maps of the Internet at the IP level are constructed by tracing routes from measurement points distributed throughout the Internet. The *skitter* system [22], which has provided data for many network topology papers, launches probes from 24 monitors towards almost a million destinations. However, a more accurate picture can potentially be built by using a larger number of vantage points. DIMES [6] heralds a new generation of large-scale systems, counting, at present 8,700 agents distributed over five continents. As some of us have pointed out [23], one of the dangers posed by a large number of monitors probing towards a common set of destinations is that the traffic may easily be mistaken for a distributed denial of service (DDoS) attack.

One way to avoid such a risk would be to avoid hitting destinations. This can be done through smart route tracing algorithms, such as Donnet et al.'s *Doubletree* [23]. With Doubletree, monitors communicate amongst themselves regarding routes that they have already traced, in order to avoid duplicating work. Since one monitor will stop tracing a route when it reaches a point that another monitor has already traced, it will not continue through to hit the destination.

Doubletree considerably reduces, but does not entirely eliminate, DDoS risk. Some monitors will continue to hit destinations, and will do so repeatedly. One way to further scale back the impact on destinations would be to introduce an additional stopping rule that requires any monitor to stop tracing when it reaches a node that is one hop before that destination. We call such a node the *penultimate node*, and we call the set of penultimate nodes the *red stop set* (RSS). Fig. 9 illustrates the RSS concept, showing penultimate nodes as grey discs.

A monitor is typically not blocked by its own first-hop node, as it will normally see a different IP address from the addresses that appear as penultimate nodes on incoming traces. This is because a router has multiple interfaces, and the IP address that is revealed is supposed to be the one that sends the probe reply. The application that we study in this paper conducts standard route traces with an RSS. We do not use Doubletree, so as to avoid having to disentangle the effects of using two different stopping rules at the same time.

How does one build the red stop set? The penultimate nodes cannot be determined a priori. However, the RSS can be constructed during a learning round in which each monitor performs a full set of standard traceroutes, i.e.,

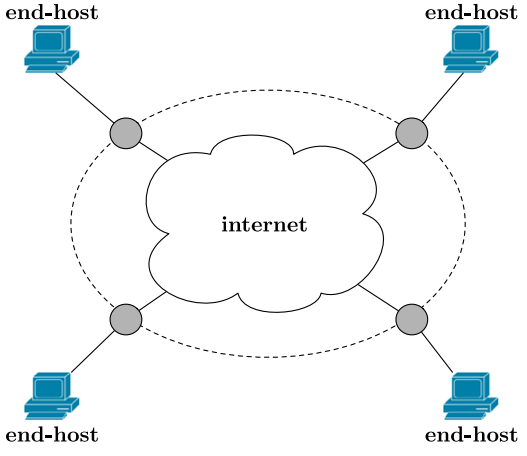

Fig. 9. Red stop set (dashed line).

Please cite this article in press as: B. Donnet et al., Improving retouched Bloom filter for trading off selected false positives against false negatives, Comput. Netw. (2010), doi:10.1016/j.comnet.2010.07.003

until hitting a destination. Monitors then share their RSSes. For simplicity, we consider that they all send their RSSes to a central server, which combines them to form a global RSS, that is then redispatched to the monitors. The monitors then apply the global RSS in a stopping rule over multiple rounds of probing.

Destinations are only hit during the learning round and as a result of errors in the probing rounds. DDoS risk diminishes with an increase in the ratio of probing rounds to learning rounds, and with a decrease in errors during the probing rounds. DDoS risk would be further reduced were we to apply Doubletree in the learning round, as the number of probes that reach destinations during the learning round would then scale less than linearly in the number of monitors. However, our focus here is on the probing rounds, which use the global RSS, and not on improving the efficiency of the learning round, which generates the RSS, and for which we already have known techniques [24].

The communication cost for sharing the RSS amongst monitors is linear in the number of monitors and in the size of the RSS representation. It is this latter size that we would like to reduce by a constant compression factor. We therefore propose encoding the RSS information in Bloom filters. Note that the central server can combine similarly constructed Bloom filters from multiple monitors, through bitwise logical OR operations, to form the filter that encodes the global RSS.

The cost of using Bloom filters is that the application will encounter false positives. A false positive, in our case study, corresponds to an early stop in the probing, i.e., before the penultimate node. We call such an error *stopping short*, and it means that part of the path that should have been discovered will go unexplored. Stopping short can also arise through network dynamics, when additional nodes are introduced, by routing changes or IP address reassignment, between the previously penultimate node and the destination. In contrast, a trace that stops at a penultimate node is deemed a *success*. A trace that hits a destination is called a *collision*. Collisions might occur because of a false negative for the penultimate node, or simply because routing dynamics have introduced a new path to the destination, and the penultimate node on that path was previously unknown.

As we show in Section 5.2, the cost of stopping short is far from negligible. If a node that has a high betweenness centrality (Dall'Asta et al. [25] point out the importance of this parameter for topology exploration) generates a false positive, then the topology information loss might be high. Consequently, our idea is to encode the RSS in an RBF.

As described earlier, there are two criteria for being able to profitably employ RBFs, and they are both met by this application. First, false positives can be identified and removed. Once the topology has been revealed, each node can be tested against the Bloom filter, and those that register positive but are not penultimate nodes are false positives. The application has the possibility of removing the most troublesome false positives by using one of the selective algorithms discussed in Section 4. Second, a low rate of false negatives is acceptable and the marginal benefit of removing the most troublesome false positives exceeds the marginal cost of introducing those false negatives. Our aim is not to eliminate collisions; if they are considerably reduced, the DDoS risk has been diminished and the RSS application can be deemed a success. On the other hand, systematically stopping short at central nodes can severely restrict topology exploration, and so we are willing to accept a low rate of random collisions in order to trace more effectively. These trade-offs are explored in Section 5.2.

5.2. Evaluation

5.2.1. Methodology

In this section, we evaluate the use of RBFs in a traceroute system based on an RSS. We first present our methodology and then, discuss our results. Note that we discuss other networking applications of RBFs in Section 6.2.

Our study is based on skitter data [22] from January 2006. This data set was generated by 24 monitors located in the United States of America, Canada, the United Kingdom, France, Sweden, the Netherlands, Japan, and New Zealand. The monitors share a common destination set of 971,080 IPv4 addresses. Each monitor cycles through the destination set at its own rate, taking typically three days to complete a cycle.

For the purpose of our study and in order to reduce computing time to a manageable level, we work from a limited set of 10 skitter monitors, all the monitors sharing a list of 10,000 destinations, randomly chosen from the original set.

We compare the following three RSS implementations: list, Bloom filter and RBF. The list would not return any errors if the network were static, however, as discussed above, network dynamics lead to a certain error rate of both collisions and instances of stopping short.

For the RBF implementation, we consider β values (see Eq. (17)) of 1%, 5%, and 25% when removing the most troublesome keys. We employ the Improved Ratio Selection algorithm, as defined in Section 4.2. For the Bloom filter and RBF implementations, the hashing is emulated with random numbers. We simulate randomness with the Mersenne Twister MT19937 pseudo-random number generator [21].

To obtain our results, we simulate one learning round on a first cycle of traceroutes from each monitor, to generate the RSS. We then simulate one probing round, using a second cycle of traceroutes. In this simulation, we replay the traceroutes, but apply the stopping rule based on the RSS, noting instances of stopping short, successes, and collisions.

5.2.2. Results

Fig. 10(a) compares the success rate, i.e., stopping at a penultimate node, of the three RSS implementations. The horizontal axis gives different filter sizes, from 10,000 to 100,000, with an increment of 10,000. Below the horizontal axis sits another axis that indicates the compression ratio of the filter, compared to the list implementation of the RSS. The vertical axis gives the success rate. A value of 0 would mean that using a particular implementation

Please cite this article in press as: B. Donnet et al., Improving retouched Bloom filter for trading off selected false positives against false negatives, Comput. Netw. (2010), doi:10.1016/j.comnet.2010.07.003

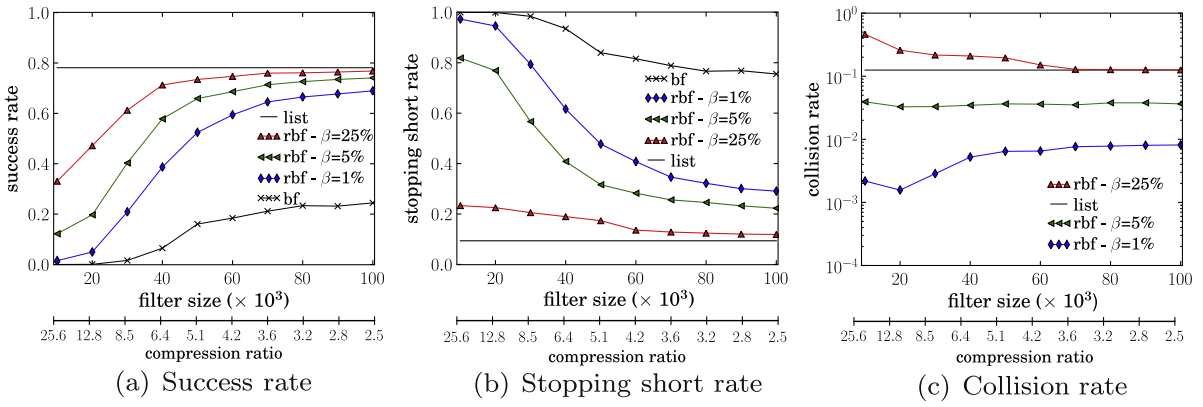

Fig. 10. RBF performance in comparison to other RSS implementations.

precludes stopping at the penultimate node. On the other hand, a value of 1 means that the implementation succeeds in stopping each time at the penultimate node.

Looking first at the list implementation (the horizontal line), we see that the list implementation success rate is not 1 but, rather, 0.7812. As explained in Section 5.2, this can be explained by the network dynamics such as routing changes and dynamic IP address allocation.

With regards to the Bloom filter implementation, we see that the results are poor. The maximum success rate, 0.2446, is obtained when the filter size is 100,000 (a compression ratio of 2.5 compared to the list). Such poor results can be explained by the troublesomeness of false positives. Fig. 11 shows, in log–log scale, the troublesomeness distribution of false positives. The horizontal axis gives the *troublesomeness degree*, defined as the number of traceroutes that stop short for a given key. The maximum value is 10^4, i.e., the total number of traceroutes performed by a monitor. The vertical axis gives the number of false positive elements having a specific troublesomeness degree. The most troublesome keys are indicated by an arrow towards the lower right of the graph: nine false positives are, each one, encountered 10,000 times.

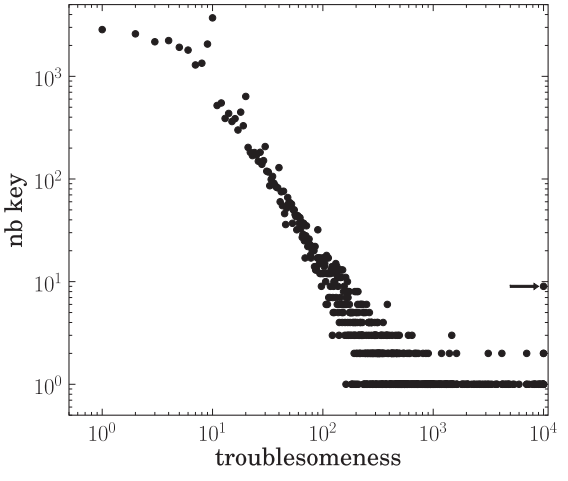

Fig. 11. Troublesomeness distribution.

We now concentrate on the RBF implementation of the RSS and, in particular, on the stopping short rate illustrated on Fig. 10(b). In this case, the advantages of an RBF over a standard Bloom filter are very interesting, in particular for quite large bit vector size and large β (i.e., $\beta = 25\%$). Indeed, for bit vector size larger than 60×10^3 and a β of 25%, the performance of an RBF is very close to the list implementation. Note again that a list might encounter stopping short due to network dynamics.

This decrease in the stopping short rate would not have any sense if the success rate when using an RBF also drops. In Fig. 10(a), we see that the RBF does better than the Bloom filter. In addition, in the fashion of the stopping short rate, the performance is close to the list implementation, particularly for large vector size and large β (i.e., $\beta = 25\%$). Indeed, when $\beta = 0.25$, for compression ratios of 4.2 and lower, the success rate approaches that of the list implementation. Even for compression ratios as high as 25.6, it is possible to have a success rate over a quarter of that offered by the list implementation.

Fig. 10(c) shows the cost in terms of collisions. Collisions could arise under Bloom filter and list implementations only due to network dynamics. One startling revelation of this figure is that for mid-range values of β, such as $\beta = 5\%$, the RBF collision cost is lower than the collision cost for the list implementation, whatever the compression ratios. On the contrary, for a large β value (i.e., 25%) and a strong compression rate, the collision cost is higher than the list implementation. This is somewhat expected as removing troublesome false positives (i.e., reducing the stopping short rate) through selective clearing when the bit vector strongly populated will generate a higher false negative rate (i.e., increasing the collision rate).

In closing, we emphasize that the construction of B and the choice of troublesome false positives is important. However, this choice is not only application-specific: the way the application considers an element as being more troublesome than another has also an impact. To leverage that choice, we must be able to associate a weight to each element. Since that instant, we are able to formulate a criterion: we want to remove $x\%$ of the whole false positives mass. From this percentage in terms of mass, we can easily

Please cite this article in press as: B. Donnet et al., Improving retouched Bloom filter for trading off selected false positives against false negatives, Comput. Netw. (2010), doi:10.1016/j.comnet.2010.07.003

deduce a percentage in terms of number. For instance, in this section, the weight associated to an IP address was its frequency and to remove 25% of the false positives mass, we had to remove only 35 keys.

6. Related work

Early suggestions of applications for Bloom filters were for dictionaries and databases. Bloom's original paper [1] describes their use for hyphenation. Another dictionary application is for spell-checkers [26,27]. For databases, they have been suggested to speed up semi-join operations [28,29] and for differential files [30,31]. Bloom filters can also find an appropriate usage in approximating membership checking of password data structures [32].

In this section, we first position our RBF regarding the state of the art (Section 6.1). We next explore networking usages in which RBF might find also a suitable usage (Section 6.2).

6.1. Others extensions of Bloom filters

There is a considerable literature on Bloom filters, and their applications in networking, that we discuss in Section 6.2. In a few instances, suggested variants on Bloom filters do allow false negatives to arise. However, these variants do not preserve the size and the membership test behavior of the standard Bloom filter, as RBFs do. Nor have the false negatives been the subject of any analytic or simulation studies. In particular, the possibility of explicitly trading off false positives for false negatives has not been studied prior to the current work, and efficient means for performing such a trade-off have not been proposed.

First is the *anti-Bloom filter*, which was suggested in unpublished work [14]. An anti-Bloom filter is composed of a standard Bloom filter plus a separate filter in which false positives from the main filter are recorded. When queried, a negative result (i.e., the key does not belong to the subset A) is generated if either the main filter does not recognize a key or the anti-filter does. The anti-Bloom filter requires more space than the standard filter, but the space efficiency has not been studied. Nor have studies been made of the impact of the anti-filter on the false positive rate. Further, it is not clear what would happen with false positives generated by the additional filter. Indeed, the anti-filter acts as a standard Bloom filter and is, thus, subject to false positives.

In some sense, the anti-Bloom filter follows the same objective as the RBF, i.e., getting rid of some of the false positives. However, the set up mechanism strongly differs. The cost of removing false positives is expressed in terms of generated false negatives in the RBF, while an anti-Bloom filter comes with additional space. Further, once the RBF is constructed, the standard behavior of a Bloom filter (for membership test) is kept, while it is not the case for anti-Bloom filter.

The objective pursued by a *Bloomier filter* [20] is not anymore to simply determine whether a given key belongs to A or not. Rather, it proposes to associate a value to each element of A and the membership test allows one to also retrieve this value (if the test is positive). A Bloomier filter is made of as many Bloom filters as different possible values. For instance, if the only permitted values are 0 and 1, a Bloomier filter uses a pair of Bloom filters S_0 and T_0 containing, respectively, all values mapping to 0 and all values mapping to 1.

When performing a membership test, if both filters replies negatively, one is sure that the key is not in the map. If one filter, let say S_0, responds positively and the other negatively, then the key is probably in S_0 and the associated value is 0. However, a problem arises when both filters respond positively. As a only single value can be associated to any key, one of the filters is lying, i.e., generating a false positive.

In such a case, one has to consider a second layer of filters, S_1 and T_1. S_1 contains values mapping to 0 and being false positives in T_0, while T_1 contains values mapping to 1 and being false positives in S_0. If this layer also produces a false positive, then one recursively applies the same scheme to a third layer, and so on.

However, if a given filter, let say S_0 replies positively to a membership query, there is still a probability that it corresponds to a false positive. This case is not addressed by the Bloomier filter while it is the core of our RBF. Further, even if a Bloomier filter is a more general Bloom filter (in the sense it allows one to associate a value to each element in the subset A), when it is reduced to a simple standard Bloom Filter (i.e., the same value is associated to each element belonging to A), it does not act as an RBF. Indeed, it does not allow one to remove selected false positives and trade off false positives and false negatives.

Fan et al.'s CBF replaces each cell of a Bloom filter's bit vector with a four-bit counter, so that instead of storing a simple 0 or a 1, the cell stores a value between 0 and 15 [3]. This additional space allows CBFs to not only encode set membership information, as standard Bloom filters do, but to also permit dynamic additions and deletions to that information. One consequence of this new flexibility is that there is a chance of generating false negatives if counters overflow. Fan et al. suggest that the counters be sized to keep the probability of false negatives to such a low threshold that they are not a factor for the application (four bits being adequate in their case). Note that recent work improves the upper bound for counter overflow probability in CBFs [33]. The possibility of trading off false positives for false negatives is not entertained.

Bonani et al.'s d-left CBF is an improvement on the CBF [17]. As with the CBF, it can produce false negatives. It can also produce another type of error called "don't know". Bonani et al. conduct experiments in which they measure the rates for the different kinds of errors, but here too there is no examination of the possibility of trading off false positives against false negatives. The d-left CBF is more space-efficient than the CBF. But CBFs themselves require a constant multiple more space than standard Bloom filters, and the question does not arise of comparing the space efficiency of d-left CBFs with that of standard Bloom filters, as they serve different functions.

Song et al. propose an extension to the CBF, called the *extended Bloom filter* (EBF), in order to support exact address prefix matching for routing [16]. An array is

Please cite this article in press as: B. Donnet et al., Improving retouched Bloom filter for trading off selected false positives against false negatives, Comput. Netw. (2010), doi:10.1016/j.comnet.2010.07.003

associated to the CBF. Each cell of this array contains the list of keys that are recorded in the corresponding cell in the CBF. Song et al. propose several techniques to reduce the memory cost of the EBFs. The EBFs are designed to achieved higher lookup performance within high-speed routers.

With the EBFs, the false positives are removed by adding information to the CBFs. With the RBFs, by contrast, no information is added to remove the false positives. The cost of these removals is expressed in terms of false negatives generated for the RBFs and in terms of increased memory usage for the EBFs.

The *variable Bloom filter* (VBF) [19] introduced by Lu et al. is basically a standard Bloom filter (i.e., an m bit vector and k hash functions) but with slightly different insertion and querying phases. During a key insertion, only t, with $t \leqslant k$, bits are set to 1 in the vector. Consequently, a key x will be declared as belonging to the filter if at least t bits amongst the k are set to 1 in the vector. In addition, a VBF allows one to remove keys from the filter. The idea is to reset d bits (with $d \leqslant t$) to zero. By acting so, a VBF is supposed to reduce the risk of false negatives when removing a key. However, Lu et al. do not provide theorical details on false positive and false negative rates of VBFs.

A VBF shares with an RBF the fact that it allows bits to be reset. However, a VBF pursues the goal of removing elements belonging to A while an RBF aims at removing troublesome false positives. Another difference with RBFs is that the resetting scheme in a VBF is less efficient than ours as it does not take into account the risk of raising false negatives.

Finally, Laufer et al. propose an extension to the standard Bloom filter called the *generalized Bloom filter* (GBF) [15]. With the GBF, one moves beyond the notion that elements must be encoded with 1s, and that 0s represent the absence of information. A GBF starts out as an arbitrary vector of both 1s and 0s, and information is encoded using two sets of hash functions, one being used for resetting bits (i.e., turning a bit to zero) and the other for setting bits (i.e., turning a bit to one). As a result, the GBF is a more general binary classifier than the standard Bloom filter. One consequence is that it can produce either false positives (as a standard Bloom filter does) and false negatives. This latter case might occur when a bit, previously set, is reset during the insertion of a subsequent element.

RBFs differ from GBFs in that the false negatives arise through the explicit removal of selected false positives. Further, an RBF, once constructed, preserves the behavior of a standard Bloom filter. We note that the techniques used to remove false positives from standard Bloom filters could be extended to remove false positives from GBFs.

6.2. Other networking usages of RBFs

Bloom filters have been widely used in networking applications, as stated by Broder and Mitzenmacher [2]. In this section, we discuss two networking usages of Bloom filters, overlays (Section 6.2.1) and packet processing (Section 6.2.2), in which RBFs might possibly find a suitable usage.

6.2.1. Overlays and peer-to-peer

For a node in a peer-to-peer file sharing system, keeping a list of objects stored at all other nodes might be costly in terms of memory, but keeping Bloom filters for all other nodes might be an attractive alternative. This was proposed by Cuenca-Acuna et al. for their *PlanetP* system [34].

PlanetP meets the two criteria for the use of RBFs. First, the application can identify false positives. A node, through is own experience with the inability to locate certain files at the expected nodes, can determine that the keys corresponding to those files yield false positives. Second, false negatives are tolerated because not every node that stores a given object need be identified. In a file sharing system, the same object is typically stored at multiple locations, and so the failure of one node to recognize some of the locations for some of the objects should not pose a great problem, provided the rate of such errors remains within reasonable bounds. The communications savings that come from eliminating some false positives might well outweigh the costs of missing some locations.

Byers et al. [35] propose an application for distributing large files to many peers in overlay networks. They suggest that peers may want to solve *approximate set reconciliation* problems. The idea is to allow a peer A to send to a peer B objects that B does not have. Encoding the sets of objects as Bloom filters allows for data compression. B will send A its Bloom filter. Testing its own set, element by element, against this Bloom filter allows A to know the set of objects B does not have, and send them to B. Because of false positives, not all objects that B needs will be sent, but most will.

Approximate set reconciliation clearly meets the second criterion for using RBFs. A low rate of false negatives in the RBF furnished by peer B would result in peer A sending a small number of elements that B already possesses. It is easy to imagine that the system designers would be willing to pay this communications overhead price in order to ensure that B gets more of the elements that it is missing.

A question arises, however, for the first criterion. How does peer B identify the false positives in the Bloom filter that it sends out? For this, it would need to know the keys for the objects that it is missing. For some applications, this would not be possible. But we could easily imagine many applications where the keys are known. For instance, B might know the contents of a music catalog, but not have many of the songs in that catalog. It could identify the false positives in its RBF by testing the keys in the catalog one by one.

Rhea and Kubiatowicz [36] describe a probabilistic algorithm for routing peer-to-peer resource location queries. Each node in the network keeps an array of Bloom filters, called an *attenuated Bloom filter*, for each adjacent edge in the overlay topology. In the array for each edge, there is a Bloom filter for each distance d, up to a maximum value, so that the dth Bloom filter in the array keeps track of resources available via d hops through the overlay network along that edge. If it is deemed probable that the resource that is being searched for is present, the query is routed to the nearest neighbor. This scheme would require the addition of feedback to identify false positives. If false positives could be identified, they could be removed. This

Please cite this article in press as: B. Donnet et al., Improving retouched Bloom filter for trading off selected false positives against false negatives, Comput. Netw. (2010), doi:10.1016/j.comnet.2010.07.003

might be worthwhile, as false negatives do not invalidate the system. The array of Bloom filters could be replaced by an array of RBFs, bringing about a decrease in the false positive rate at the cost of a comparatively small increase in the false negative rate.

6.2.2. Network packet processing

Dharmapurikar et al. [37] propose the use of Bloom filters for detecting predefined signatures in packet payloads. They propose an architecture of *W parallel Bloom filters*, each Bloom filter focusing on strings of a specified length. If a string is found to be a member of any Bloom filter, it is then declared as a possible matching signature. To avoid the risk of false positives, each matching signature is tested in an *analyzer* which determines if the signature is truly a member of the set *A* or not. In other words, the analyzer contains all elements of *A*. Bloom filters are only used to discard elements not belonging to *A*.

At least one of the two criteria for using the RBFs is met in the process described by Dharmapurikar et al. The analyzer offers the opportunity to identify false positives. The application should obtain a gain in terms of processing time by removing from the filters those false positives. The second criterion is application-specific. If a small rate of false negatives may be tolerated, then RBFs are suitable.

7. Conclusion

The Bloom filter is a lossy summary technique for testing set membership. It has attracted considerable attention from the networking research community for its bandwidth efficiencies. This paper has described the retouched Bloom filter (RBF), an extension that makes Bloom filters more flexible by permitting selected false positives to be removed at the expense of introducing some false negatives. The key idea is to remove each false positive by resetting a carefully chosen bit in the bit vector that makes up the filter. Compared to a standard Bloom filter, an RBF introduces a modification in the filter construction, but keeps intact both the bit vector size and the membership test procedure.

We analytically demonstrated that the trade-off between false positives and false negatives is at worst neutral, on average, when randomly resetting bits in the bit vector, whether these bits correspond to false positives or not. We also proposed three algorithms for deleting false positives. The costs associated with these algorithms are constant multiples of the spatial and time complexity required to build a standard Bloom filter. We evaluated these algorithms through simulation and showed that RBFs created in this manner will, on average, increase the false negative rate by less than the amount by which the false positive rate is decreased. The algorithms in this paper are more effective than similar algorithms that we presented in an earlier version of this work [4].

This paper described a distributed network measurement application for which RBFs can profitably be used. In this case study, route tracing agents, rather than conducting traceroutes all the way to each destination, attempt to terminate each measurement at the penultimate node, in this way reducing their impact on end-hosts and reducing the likelihood that measurements will trigger intrusion detection alerts. The agents share information on the set of penultimate nodes that each has discovered. We compared three different implementations for representing this set information: list, Bloom filter, and RBF. Using filters reduces the bandwidth requirements, but the false positives can significantly reduce the amount of topology information that the system gleans. We demonstrated that using an RBF, in which the most troublesome false positives are removed, will increase the coverage compared to a standard Bloom filter implementation, while retaining the bandwidth efficiencies. The RBF implementation is less effective than the standard Bloom filter implementation at protecting destinations. However, as we note, the application needs only to reduce impact, not eliminate it entirely, and the RBF enables this.

In future work, we hope to demonstrate techniques to apply the RBF concept earlier in the construction of the filter. At present, we allow the Bloom filter to be built, and then remove the most troublesome false positives. It should be possible to avoid recording some of these false positives in the filter to begin with. We also aim at modeling a notion of mass associated with elements stored in the filter in order to better apply the concepts discussed in this paper.

Acknowledgements

Mr. Donnet's work is supported by the FNRS (Fonds National de la Recherche Scientifique, rue d'Egmont 5 – 1000 Bruxelles, Belgium). Mr. Friedman's work has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement No. 224263-OneLab2.

The authors thank Mark Crovella who introduced them to Bloom filters and encouraged their work. Authors thank kc claffy and her team at CAIDA for allowing them to use the skitter data. Rafael P. Laufer suggested useful references regarding Bloom filter variants. Otto Carlos M.B. Duarte helped us clarify the relationship of RBFs to such variants.

References

[1] B.H. Bloom, Space/time trade-offs in hash coding with allowable errors, Communications of the ACM 13 (7) (1970) 422–426.
[2] A. Broder, M. Mitzenmacher, Network applications of Bloom filters: a survey, Internet Mathematics 1 (4) (2002) 485–509.
[3] L. Fan, P. Cao, J. Almeida, A.Z. Broder, Summary cache: a scalable wide-area Web cache sharing protocol, IEEE/ACM Transactions on Networking 8 (3) (2000) 281–293.
[4] B. Donnet, B. Baynat, T. Friedman, Retouched Bloom filters: allowing networked applications to trade off selected false positives against false negatives, in: Proceedings of the ACM SIGCOMM CoNEXT, 2006.
[5] k. claffy, Y. Hyun, K. Keys, M. Fomenkov, D. Krioukov, Internet mapping: from art to science, in: Proceedings of the IEEE Cybersecurity Applications and Technologies Conference for Homeland Security (CATCH), 2009.
[6] Y. Shavitt, E. Shir, DIMES: let the internet measure itself, ACM SIGCOMM Computer Communication Review 35 (5) (2005) 71–74.
[7] F. Georgatos, F. Gruber, D. Karrenberg, M. Santcroos, A. Susanj, H. Uijterwaal, R. Wilhelm, Providing active measurements as a regular service for ISPs, in: Proceedings of the Passive and Active Measurement Workshop (PAM), 2001.
[8] H.V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, A. Venkataramani, iPlane: an information plane for

distributed services, in: Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2006.
[9] Wide, Gulliver: distributed active measurement project, 2006. <http://gulliver.wide.ad.jp/>.
[10] K. Chen, D. Choffnes, R. Potharaju, Y. Chen, F. Bustamante, D. Pei, Y. Zhao, Where the sidewalk ends: extending the Internet AS graph using traceroutes from P2P users, in: Proceedings of the ACM SIGCOMM CoNEXT, 2009.
[11] T. Bourgeau, J. Augé, T. Friedman, TopHat: supporting experiments through measurement infrastructure federation, in: Proceedings of the International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom), 2010.
[12] k. claffy, CAIDA 2010–2013 program plan, 2010. <http://www.caida.org/home/about/progplan/progplan2010/>.
[13] D.R. Choffnes, F.E. Bustamante, Taming the torrent: a practical approach to reducing cross-ISP traffic in P2P systems, in: Proceedings of the ACM SIGCOMM, 2008.
[14] T. Lavian, Bloom filters, CS 270 – class notes, 2004. <http://www.cs.berkeley.edu/kamalika/cs270/notes/lecture30b.pdf>.
[15] R.P. Laufer, P.B. Velloso, D. de O. Cunha, I.M. Moraes, M.D.D. Bicudo, O.C.M.B. Duarte, A new IP traceback system against distributed denial-of-service attacks, in: Proceedings of the 12th International Conference on Telecommunications (ICT), 2005.
[16] H. Song, S. Dharmapurikar, J. Turner, J. Lockwood, Fast hash table lookup using extended Bloom filter: an aid to network processing, in: Proceedings of the ACM SIGCOMM, 2005.
[17] F. Bonomi, M. Mitzenmacher, R. Panigraphy, S. Singh, G. Varghese, Beyond Bloom filters: from approximate membership checks to approximate state machines, in: Proceedings of the ACM SIGCOMM, 2006.
[18] A. Kirsch, M. Mitzenmacher, Distance-sensitive Bloom filters, in: Proceedings of the Algorithm Engineering and Experiments (ALENEX), 2006.
[19] Y. Lu, B. Prabhakar, F. Bonomi, Bloom filters: design, innovations, and novel applications, in: Proceedings of the 43rd Annual Allerton Conference, 2005.
[20] B. Chazelle, J. Killian, R. Rubinfeld, A. Tal, The Bloomier filter: an efficient data structure for static support lookup tables, in: Proceedings of the 15th ACM/SIAM Symposium on Discrete Algorithms (SODA), 2004.
[21] M. Matsumoto, T. Nishimura, Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator, ACM Transactions on Modeling and Computer Simulation 8 (1) (1998) 3–30.
[22] B. Huffaker, D. Plummer, D. Moore, k. claffy, Topology discovery by active probing, in: Proceedings of the Symposium on Applications and the Internet (SAINT), 2002.
[23] B. Donnet, P. Raoult, T. Friedman, Deployment of a large-scale topology discovery algorithm, IEEE Journal on Selected Areas in Communications, Sampling the Internet: Techniques and Applications 24 (12) (2006) 2210–2220.
[24] B. Donnet, T. Friedman, Internet topology discovery: a survey, IEEE Communications Survey and Tutorials 9 (4) (2007) 2–15.
[25] L. Dall'Asta, I. Alvarez-Hamelin, A. Barrat, A. Vázquez, A. Vespignani, A statistical approach to the traceroute-like exploration of networks: theory and simulations, in: Proceedings of the Combinatorial and Algorithmic Aspects of Networking Workshop (CAAN), 2004.
[26] M.D. McIlroy, Development of a spelling list, IEEE Transactions on Communications 30 (1) (1982) 91–99.
[27] J.K. Mullin, D.J. Margoliash, A tale of three spelling checkers, Software – Practice and Experience 20 (6) (1990) 625–630.
[28] K. Bratbergsengen, Hashing methods and relational algebra operations, in: Proceedings of the 10th International Conference on Very Large Databases, 1984.
[29] P. Valdurez, G. Gardarin, Join and semijoin algorithms for a multiprocessor database machine, ACM Transactions on Database Systems 9 (1) (1984) 133–161.
[30] L.L. Gremilion, Designing a Bloom filter for differential file access, Communications of the ACM 25 (9) (1982) 600–604.
[31] J.K. Mullin, A second look at Bloom filters, Communications of the ACM 26 (8) (1983) 570–571.
[32] U. Mamber, S. Wu, An algorithm for approximate membership checking with application to password security, Information Processing Letters 50 (4) (1994) 191–197.
[33] D. Ficara, S. Giordano, G. Procissi, F. Vitucci, Multilayer compressed counting Bloom filters, in: Proceedings of the IEEE INFOCOM, 2008.
[34] F.M. Cuenca-Acuna, C. Peery, R.P. Martin, T.D. Nguyen, PlanetP: using gossiping to build content addressable peer-to-peer information sharing communities, in: Proceedings of the 12th IEEE Symposium on High Performance Distributed Computing (HPDC), 2003.
[35] J. Byers, J. Considine, M. Mitzenmacher, S. Rost, Informed content delivery over adaptive overlay networks, in: Proceedings of the ACM SIGCOMM, 2002.
[36] S.C. Rhea, J. Kubiatowicz, Probabilistic location and routing, in: Proceedings of the IEEE INFOCOM, 2002.
[37] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, J. Lockwood, Deep packet inspection using parallel Bloom filters, IEEE Micro 24 (1) (2004) 52–61.

Benoit Donnet received his M.S. degree in computer science from the Institut d'Informatique of the Facultés Universitaires Notre Dame De La Paix (Namur – Belgium) in 2003. Mr. Donnet received his Ph.D. degree in computer science form the Université Pierre et Marie Curie in 2006. He is currently FNRS fellow at the Université catholique de Louvain in the IP Networking Lab (http://inl.info.ucl.ac.be). His research interests are in Internet measurements, focusing on scalable measurement techniques, Bloom filters, and traffic engineering techniques.

Bruno Baynat received his M.S. degree from the Institut National Polytechnique de Grenoble (INPG) in 1988 and the Ph.D. degree in computer science from the Université Pierre et Marie Curie in 1991. He is currently a Maître de Conférences (assistant professor) of computer science at the Université Pierre et Marie Curie in Paris, and a researcher at the Laboratoire d'Informatique de Paris 6 (LIP6). His research interests are presently in the development of analytical models for the performance evaluation of communication systems, with applications to wired networks (Multicast, metrology of IP networks) and wireless networks (Ad-Hoc, GPRS/EDGE/UMTS, WI-FI).

Timur Friedman received the A.B. degree in philosophy from Harvard University and the M.S. degree in management from Stevens Institute of Technology. He received the M.S. and Ph.D. degrees in computer science from the University of Massachusetts Amherst, in 1995 and 2001, respectively. He is currently a Maître de Conférences (assistant professor) of computer science at the Université Pierre et Marie Curie in Paris, and a researcher at the Laboratoire d'Informatique de Paris 6 (LIP6). His research interests include large-scale network measurement systems and disruption tolerant networking.

Please cite this article in press as: B. Donnet et al., Improving retouched Bloom filter for trading off selected false positives against false negatives, Comput. Netw. (2010), doi:10.1016/j.comnet.2010.07.003