Folding and Unfolding Bloom Filters - An off-Line Planing and on-Line Optimisation

Francoise Sailhan*†, and Mark-Oliver Stehr*
* Computer Science Laboratory, SRI international
Menlo Park, CA 94025 USA
Email: firstName.lastName@csl.sri.com
†Cedric Laboratory, CNAM, 75003 Paris, France
Email: firstName.lastName@cnam.fr

I. Introduction

Last decade is characterised by the convergence of pervasive technologies including wireless communication, smart devices and the Internet. The resulting Internet of the things enables not-only users but also things to access resource anywhere, anytime. In practice, data is ubiquitously flowed relying on global wireless connectivity (e.g., 3G) or on *ad hoc*-based local area networking, as enabled by e.g., IEEE 802.15.4 protocols. One key limitation of those networks remains the limited bandwidth and energy. In this context, early-stage data and resource discovery should be provided so as to provide efficient data dissemination [12]. A Bloom filter is a space-efficient structure (namely a vector of bits) that is dedicated to the representation of a set and designed so as to support efficient membership queries. It distinguished itself by the fact that the time associated with querying a membership is independent of the number of elements stored in the Bloom filter as well the size of the Bloom filter. In a nutshell, the Bloom filter works as follows. An element intended to be added to the Bloom filter is first hashed. Then, the output is used to flip to 1 the bit at the related position in the Bloom filter. Typically, $k > 1$ hash functions are used. In order to check whether an element is stored in the Bloom filter, the queried element is hashed and if the bits at the related position are set to 1 in the Bloom filter, then, this element is stored. It is worth to mention that false negatives are not encountered. Nevertheless, the approximate nature of Bloom filter implies that false positives might take place. The probability of false positive depends of (i) the Bloom filter size, (ii) the number of hash functions and (iii) the number of elements that are stored. Keeping to a minimum the false positive rate requires the dynamically adaptation of the number of hash functions or/and the Bloom filter size. Further attempts to increase the size of the Bloom filter appeared in the literature. The basic idea consists in creating a set of Bloom filters [5], [15], [1]. As a side effect, a membership request becomes a function of the size of this set. While these methods deal with the increase of the Bloom filter size, the related decrease is neglected. An alternative [6] consists in partitioning the Bloom filter in k disjoint sub-ranges wherein an element that is added by hashing this latter with the related hash function h_k. Depending on the expected probability of false positive, k is either dynamically increased and also decreased, hence leading to an reduced/increased Bloom filter size. Nevertheless, still changing the number of hash functions offers a lower granularity on the resizing of the Bloom filter comparing to a direct fine-grained resizing.

We herein consider the problem of dynamically resizing the Bloom filter by folding and unfolding it. This can be expressed as follows. Given a number of items stored in the Bloom filter, let us reduce/ increase the size of the Bloom filter by folding/unfolding it so that the false positive rate keeps into $[\rho - \epsilon, \rho + \epsilon]$ and that the unnecessary increase of the memory size is kept to a minimum. We acknowledge that the halve of a Bloom filter has been originally suggested in [13] and successfully applied [4] so as to reduce the bandwidth consumption induced by the exchange of Bloom filters in the context of correlated anomaly detection in large-scale Grid [14]. We herein extend/generalize this approach by introducing the concept of folded/unfolded Bloom filter, which permits a highly flexible resize. We also introduce a novel formulation of the problem. The key challenge consists in determining how a folding should be performed, namely the number of times the Bloom filter should be folded/unfolded and the reduction factor associated with each fold/ unfold. We show that this can be formulated as (i) an off-line planing of the factorization of an integer (namely the Bloom filter size) and (ii) an on-line optimization of the Bloom filter folding/unfolding. Our main contribution is twofolds. We propose an algorithm for planing the fold and unfold of the bloom filter depending on the expected false positive rate and the related resource waste. We formulate this problem and provide both a theoretical and experimental analysis of such planing. In addition, Based on this planed folds/unfolds, we propose an approach to dynamically optimize the fold and unfold of a Bloom filter.

The remaining is organized as follows. We first introduce Bloom filters and survey the related literature (§II). Then, we present the Bloom filter fold/unfold (§III) and the planing of the fold/unfold (§IV. Finally, we conclude this report with further optimization and research directions (§V).

This work carried when F. Sailhan was a visiting fellow at SRI.

II. Background on Bloom filters

Bloom filters [2] are commonly used to represent a set so as to support efficient membership queries, that is to efficiently test whether an element is a member of a set. For this purpose, a Bloom filter B is represented as a vector of bits, denoted $b(1), \cdots, b(m)$. This vector is initially set to 0, i.e., $\forall i \in [0, m]$, $b(i) = 0$ and updated by adding additional elements.

a) Bloom Filter Update: in order to add an element e into the Bloom filter, k hash functions $h_1, \cdots, h_k$ are used. Each hash function is applied to the element: $h_1(e), \cdots, h_k(e)$. Then, the k bits at positions $h_1(e) \ mod(m), \cdots, h_k(e) \ mod(m)$ are set to 1.

b) Checking the membership of an element: the process of checking whether an element q belongs to the Bloom filter is very similar to adding an element. First, q is hashed: $h_1(q) \ mod(m), \cdots, h_k(q) \ mod(m)$. If one bit at the related positions, is set to 0, then the element is not stored in the Bloom filter. Otherwise (i.e., if none of those bits is set to 0), the element is said to be stored in the Bloom filter.

c) Suppressing an element. : items cannot be removed from a Bloom filter: deleting an element in the Bloom filter cannot be handled by flipping a bit back to 0. In order to deal with this issue, few proposals emerged. The so-called counting Bloom filter [3] is the most populare. A counting vector $c_1, \cdots, c_m$ replaces the bit vector of the Bloom filter. It records the number of items stored in the Bloom filter. The removal of an element e is handled by decrementing the k counters situated at $h_1(e), \cdots, h_k(e)$. In counterpart, the size of the counting vector is greater than the Bloom filter size by a factor that depends of the space allocated to count. These counters should be selected large enough so as to avoid overflows.

A. Properties of Bloom Filters

A (counting) Bloom filter is characterized by the following properties:

1) the time associated with a membership query is independent of the number of elements stored in the (counting) Bloom filter as well as its size. More precisely, the time dedicated to insert/search an element depends on the number of hash functions and is hence $o(k)$,
2) Both basic and counting Bloom filters encounter fault positive. Given n the number of elements that are stored in the (counting) Bloom filter (see Table I) and supposing that the hash function are perfect, the probability of a false positive, denoted ρ, verifies:

$$\rho = (1 - (1 - \frac{1}{m})^{kn})^k \approx (1 - e^{\frac{kn}{m}}) \qquad (1)$$

   Contrary to basic Bloom filter, a counting Bloom filter may encounter a fault negative which occurs if the counting vector is undersized and meets an overflow.
3) The union of two sets S_1 and S_2 can be obtained as follows:
   - applying a bit wise OR on the two corresponding basic Bloom filters.
   - adding the two counting Bloom filters. Note that the resulting value cannot exceed a maximum value defined by the counters sizes.
4) The intersection of two sets S_1 and S_2 cannot be obtained in a deterministic manner on the (counting) Bloom filter,
5) With counting Bloom filter, the subtraction of two sets $S_1 - S_2$ can be obtained by subtracting the corresponding counting Bloom filters, $CB_1 - CB_2$. Basic Bloom filters do not support such subtracting.

Despite the above promising properties, (counting) Bloom filter suffers from a fundamental shortcoming: once set, the size of the Bloom filter cannot be easily changed. An over- or under-estimate of this size leads to a waste of memory/bandwidth or/and an increase of the false positive. Few approaches have been proposed to deal with this issue.

B. Related Work

In order to control the false positive rate of a Bloom filter, one may change the size of the Bloom filter or the number of hash functions. The approaches proposed so as to increase the Bloom filter size, are slightly different [5], [15], [1]. All consist in using a set of Bloom filters, i.e., a matrix of Bloom filters. Briefly sketched, if the false positive rate (or if the number of items stored) exceeds a given threshold, then another Bloom filter is added to this matrix. Proposed approaches differ in the size of the Bloom filter. In [5], the Bloom filters are of equal size: a matrix $m \times s$ is handled, with $s = \lceil \frac{n}{n_0} \rceil$ and n defining the actual number of items added to the matrix and n_0 denoting the number of items that can be added to a single Bloom filter. The time associated with inserting an element (in one of the s Bloom filters) remains the same as with basic Bloom filters (namely $o(k)$). Nevertheless, the time associated with searching an element increases and reaches in the worthies: $o(k \times \frac{\lceil \frac{n}{n_0} \rceil + 2}{2})$. Similar approaches are proposed in [15], [1], except that the size of the added Bloom filters follows an exponential grows with the former (i.e., if $n > 2^{i-1} n_0$ then $m = i \times m_0$), and that Bloom filters are added so that the probability of false positive follows a geometric law $p_o, p_0.r, p_o r^2$ for the latter. Together, these approaches imply an increase of the time devoted to look for an item. This increase is characterized by a factor s. In addition, although the Bloom filter size can be increased, the decrease is not addressed and cannot be addressed in a simple manner. Such a decrease could be provided through compression [7]. It is worth to mention that counting Bloom filter could not be easily applied here. An alternative to increasing the (counting) Bloom filter size, lies in changing the number of hash functions [6]. This approach also called the partitioned (counting) Bloom filter, lies in allocating the k hash functions to disjoint $\frac{m}{k}$ ranges in the Bloom filter. This permits to easily increase or decrease the (counting) Bloom filter size, as needed. In addition, the adding of an item can be parallelised and the asymptotic performance of a partitioned Bloom filter remains the same as with Bloom filter. Nevertheless, performance of a partitioned Bloom filter are poorer due to the $\frac{m}{k}$ restriction.

m:	Bloom filter size
$b(1), \cdots, b(m)$:	Bloom filter
k:	number of hash functions
$h_1(), \cdots, h_k()$:	set of hash functions
n:	set of items stored in the Bloom filter

TABLE I
NOTATION

In order to tackle this issue, we propose to adapt the Bloom filter size by dynamically folding and unfolding it. As pointed out in [13], promoted and successfully applied in [4], a nice feature of Bloom filters is that they can be halved in size, supposing that the size of the filter is a power of 2. In order to halve a filter, an OR (resp. addition) on the first and second halves of the Bloom filter (resp. counting Bloom filter) is performed. This approach has been successfully exploited to reduce the bandwidth allocated to transmit a Bloom filter.

Going one step further, we generalize this approach by:

- introducing the notion of fold and unfold of a (counting) Bloom filter, which permits to increase or decrease the Bloom filter size in a flexible manner,
- providing a formulation of the problem related to the planing of the folds and unfolds, while proposing an analytical and experimental evaluation of this planning,
- formally describing the process of fold and unfold and proposing further approaches for optimizing these operations.

III. Dynamic Bloom filter

In order to adapt dynamically the Bloom filter size, we propose to fold and respectively unfold the Bloom filter when the number of elements stored in the Bloom filter decreases or respectively increases. We shall introduce this fold and unfold formally, but first let us describe the intuitive idea behind these operations. A fold (Figure 1) can be metaphorically illustrated relying on a piece of paper that is folded ; the piece of paper represents the Bloom filter. A paper of 50cm can be folded into a paper of 10cm (size divided by 5). Then, it can be folded into a paper of 5cm (size divided by 2). This can be again folded into a paper of size 1cm (size reduced by 5). Such a fold corresponds to a so-called *flat fold* wherein the piece of paper is folded back into the plan so that the paper necessarily touches itself. This fold is characterized by several properties:

- we are concerned with simple and flat folding in one dimension, i.e., each fold takes a flat piece of paper and fold it in another flat piece of paper by a rotation by 180°. It follows that the paper cannot self-cross and multiple layers of papers resulting from (successive) folds may touch,
- the reduced size of the Bloom filter remains an integer. This is necessary so that a reduced Bloom filter remains a Bloom filter (an array of bits, whose size is an integer) after the reduction,
- folding are of equal size. This is required to enforce an equi-probable repartition of the bits in the resulting (folded, counting) Bloom filter.

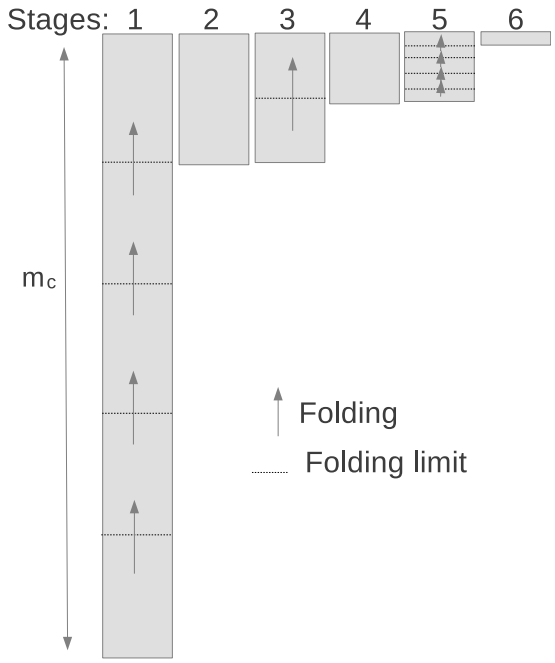

Fig. 1. Fold

- a folding pattern is made of a collection of foldings. The order of the foldings materialised by the overlap order and the top-bottom orientation, have not impact on the resulting folded Bloom filter.
- A folding is characterized by:
  - *reduction factor* that reflects the ratio of reduction of the Bloom filter that is folded. For instance, the reduction factor 5 is obtained at stage 1 (Figure 1), i.e., when the Bloom filter of 50 elements is folded into a Bloom filter of 10 elements.
  - *folding rank* that defines the number of times such a folding is performed. In the example provided in Figure 1, a reduction factor is applied 2 times. The rank of the reduction factor 5 is hence equal to 2.

  The reduction factor corresponds to a multiplicative factor of the Bloom filter size. For instance (Figure 1), the Bloom filter is folded/divided by the following factors: 5 (stage 1), 2 (stage 3), 5 (stage 5). These factors multiplied together form m, the original size of the Bloom filter:

$50 = 5^2.2$. Note also that 5 and 2 are primes.

Note that the same folding applies to a counting Bloom filter. Unless especially pointed out, the term Bloom filter will hereafter encompass the notion of basic and counting Bloom filter together.

A. Fold Planing

Let us define in a formal way the folding and unfolding of a Bloom filter. Initially (i.e., at stage 1), the Bloom filter is oversized ; its size is set to its maximal capacity denoted m_1. Once set, the Bloom filter size can be dynamically adjusted within $[1, m_1]$. Recall that any positive integer (including m_1) can be represented as a unique product of power of primes, the canonical factorization of m_1 is henceforth of the following form:

$$m_1 = \prod_{j=1}^{a} (p_j)^{\alpha_j} \quad \text{with} \quad a \gg 1. \tag{2}$$

In order to warrant that the Bloom filter is highly foldable, the number m_1 should be selected so that (ii) it corresponds to a composite number (rather than a prime) and (ii) there exists many possible foldings, i.e., $a \gg 1$ (see §IV-A for a detailed explanation on the selection of m_1). Then, once the size is set, the Bloom filter can be folded. There exists three ways of folding a Bloom filter:

- *an elementary folding* corresponds to a single time folding characterised by a "minimal" factor of reduction corresponding to a prime. For instance (Figure 1), at stage 1, the elementary folding of the Bloom filter follows a factor reduction of 5.
- a *composite folding* that results from a single time characterised by a factor of reduction corresponding to a composite number (i.e., not a prime). For instance, a Bloom filter size can be reducted by a factor of 4.
- a *sequencial* folding corresponding to a succession of elementary and/or composite foldings. in Figure 1, a sequence of elementary folding is furnished.

Elementary fold - We consider the elementary fold of a Bloom filter B_t, which results into a folded Bloom filter denoted B_{t+1}. Let p_l be the reduction factor of this initial fold and ϕ_{p_l} represent the folding function. This reduction factor can be expressed as the following quotient: $p_l = \frac{m_t}{m_{t+1}}$ with m_t (resp. m_{t+1}) corresponding to the size of B_t (resp. B_{t+1}) and $l \in [0, a]$. Naturally, the folding differs depending on the Bloom filter that is considered:

- A basic Bloom filter is folded by relying on the logical OR operator. More precisely, this consists in operating a OR on the Bloom filter portions that are folded,
- A counting Bloom filter is folded by adding the different parts of the counting Bloom filter that is folded.

In order to describe in an unique manner the folding operation that takes place on a basic/counting Bloom filter, we hereafter utilize the notation $\oplus$ for both OR and sum. In addition, we utilise the notation B so as to represent a basic or counting Bloom filter. Overall, the folding of a Bloom filter is performed as follows:

$$\forall i \in [1, m_{t+1}], b_{t+1}(i) = \oplus_{l=0}^{p_l - 1} b_t(i + l.m_{t+1}) \tag{3}$$

Thus, the resulting Bloom filter can be expressed as:

$$B_{t+1} = \begin{pmatrix} \oplus_{l=0}^{p_l-1} b_t(1 + l.m_{t+1}) \\ \cdots \\ \oplus_{l=0}^{p_l-1} b_t(i + l.m_{t+1}) \\ \cdots \\ \oplus_{l=0}^{p_l-1} b_t(m_{t+1} + l.m_{t+1}) \end{pmatrix}$$

$$\text{with} \quad B_t = \begin{pmatrix} b_t(1) \\ \cdots \\ b_t(i) \\ \cdots \\ b_t(m_t) \end{pmatrix}$$

The computing cost associated with this elementary folding is $o(p_l)$, i.e., it depends of the size of the portions that are folded. The probability of false positive ρ on the folded Bloom filter B_{t+1} containing n elements, using k hashing functions verifies:

$$\rho = \rho(m_{t+1}, n, k) = (1 - (1 - \frac{1}{m_{t+1}})^{k.n})^k \approx (1 - e^{\frac{k.n}{m}}) \tag{4}$$

In other words, the probability of false positive occuring on a folded Bloom filter is the same as with a Bloom filter which has the same size. Overall, such an elementary folding is intented to be composed or/and sequentially performed so as to further reduce the Bloom filter size.

Composite and Sequential Folding - Composite and sequencial folds behave differently. Whereas the composite folding is performed in a single time relying on a high reduction factor, a sequential folding is operated by successively applying either an elementary or/and composite folding. Despite this difference, the Bloom filters resulting of a composite folding ($\phi_{p_r.p_l}$) and sequential folding (one time folding $\phi_{p_r}.\phi_{p_l}$) is identical.

Lemma - The sequential composition, denoted o, of 2 elementary foldings ϕ_{p_l} and ϕ_{p_r} verifies:

$$\phi_{p_l} o\, \phi_{p_r} = \phi_{p_r.p_l} \tag{5}$$

Proof: Given a Bloom filter B_t, let us proove that:

$$\phi_{p_l}\, o\, \phi_{p_r}(B_t) = \phi_{p_l.p_r}(B_t)$$

The successive folding of B_t leading to B_{t+1}, and then the folding of B_{t+1} resulting in B_{t+2} can be expressed as:

$$\forall i \in [1, m_{t+2}], b_{t+2} = \phi_{p_l}\, o\, \phi_{p_r}\Big(b_t(i)\Big) = \phi_{p_l}\Big(b_{t+1}(i)\Big)$$

$$\text{with } \forall i \in [1, m_{t+1}], b_{t+1}(i) = \oplus_{r=0}^{p_r-1} b_t(i + r.m_{t+1})$$

$$\Rightarrow \forall i \in [1, m_{t+2}], b_{t+2}(i) = \oplus_{l=0}^{p_l-1} \oplus_{r=0}^{p_r-1} b_t(i + l.m_{t+2} + r.m_{t+1})$$

$$= \oplus_{l=0}^{p_l-1} \oplus_{r=0}^{p_r-1} b_t(i + l.\frac{m_t}{p_l.p_r} + r.\frac{m_t}{p_r})$$

Meanwhile, a one-time-composite folding of B_t which leads to B_{t+2} is performed as follows:

$$\forall i \in [1, m_{t+2}], b_{t+2}(i) = \phi_{p_r.p_l}(b_t(i)) = \oplus_{j=0}^{p_r.p_l-1} b_t(i + j.m_{t+2}) = \oplus_{j=0}^{p_r.p_l-1} b_t(i + j.\frac{m_t}{p_r.p_l})$$

It follows that $\phi_{p_l}\ o\ \phi_{p_r}(b_t) = \phi_{p_r.p_l}(b_t)$ given $l \in [0, p_l]$, $r \in [0, p_r]$, and, $j = l + rp_l$ $\square$. Although a sequential folding $\phi_{p_r} o \phi_{p_l}$ and a one time folding $\phi_{p_r.p_l}$ provide the same result, the cost in terms of time, resulting from a sequential folding is greater than the cost of a composite folding. More particularly, sequential folding is $o\Big(m_t.(1+\frac{1}{p_l})\Big)$ and a composite folding $\phi_{p_r.p_l}$ is $o(m_t)$.

Theorem - The sequential folding, denoted o, of 2 elementary foldings ϕ_{p_l} and ϕ_{p_r} is:

- commutative, i.e., $\phi_{p_l}\ o\ \phi_{p_r} = \phi_{p_r}\ o\ \phi_{p_l}$
- associative, i.e., $\phi_{p_l} o\ (\phi_{p_r}\ o\ \phi_{p_u}) = (\phi_{p_r}\ o\ \phi_{p_l})\ o\ \phi_{p_u}$

Proof:

$\phi_{p_r}\ o\ \phi_{p_l}(B_t) = \phi_{p_l.p_r}(B_t) = \phi_{p_r.p_l}(B_t) = \phi_{p_l}\ o\ \phi_{p_r}(B_t)$.
Thus, $\phi_{p_l}\ o\ \phi_{p_r} = \phi_{p_r}\ o\ \phi_{p_l}$ $\square$
In addition, $\phi_{p_l} o\ (\phi_{p_r}\ o\ \phi_{p_u}) = \phi_{p_r.p_l.p_u} = (\phi_{p_r}\ o\ \phi_{p_l})\ o\ \phi_{p_u}$
Thus, $\phi_{p_l} o\ (\phi_{p_r}\ o\ \phi_{p_u}) = \phi_{p_l} o\ (\phi_{p_r}\ o\ \phi_{p_u})$ $\square$

The commutative property of the sequential folding implies that the order of the folding does not count. More generally, given a folded Bloom filter B_t, it is impossible to guess whether the Bloom filter went through an elementary, sequential or composite folds : the folding pathway cannot be inspected given the resulting Bloom filter. Until now, we have focus our study on the folding. Let us analyze the reverse operation.

Unfold. With Bloom filter, the unfold cannot be provided as a simple operation e.g., a AND or a substraction. Let B_t represent a Bloom filter (at stage t) that is folded resulting in B_{t+1}. Assuming that B_t results from f foldings (with $f < t$), i.e., $B_t = \phi_{g_{j_1}} o \cdots o \phi_{g_{j_f}}(B_1)$ with g corresponding to either a prime or composite number (see Section IV-A). The unfolding of B_{t+1} into B_t is obtained by regenerating B_t based on the original Bloom filter B_1:

$$\text{Given } B_{t+1} = \phi_{g_{j_{f+1}}}\Big(B_t\Big),\ B_t = \phi^{-1}_{g_{j_{f+1}}}\Big(B_{t+1}\Big) = \phi_{g_{j_1} \ldots g_{j_f}}\Big(B_1\Big)$$

It follows that the cost associated with this process corresponds to the cost associated with a composite folding.

Membership Query - Querying if an item q belongs to a (counting, folded) Bloom filter is straightforward. Let assume that f foldings denoted $j_1, \cdots, j_f$. The membership query proceeds as follows. First, q is hashed with the k hash functions. Then, if one of the k investigated bits is set to 0, then the element q is not stored. Note that given that the Bloom filter has been folded, the position of the investigated bit should be proportionally divided. In practice, such a division consists in taking the modulo, denoted mod, of the hashed item $h_k(q)$. Thus, the bits that are investigated are those that are located at position $mod\big(\cdots mod\big(mod\big(h_k(q), p_{j_1}\big), p_{j_2}\big), \cdots, p_{j_f}\big)$ are investigated. More formally, q is not stored iff:

$$\exists k \ni: B\Big[mod\big(h_k(q), \prod_{i=0}^{f}(p_{j_i})\big)\Big] = 0 \quad (6)$$

The cost associated with a query membership is $o(k)$. Also, the addition of an item or the removal of an item (in the case of a counting Bloom filter) is performed similarly and leads to an $o(k)$ operation.

Looking Back - A folded Bloom filter results from a sequential composition of elementary or composite folding. Although sequential folding $\phi_{p_r}\ o\ \phi_{p_l}$ and composite folding $\phi_{p_r.p_l}$ provide the same result, the cost in terms of time, resulting from a sequential folding is greater than the cost of a one time folding. As summarized in Table II, folding or unfolding a Bloom filter implies an additional cost that depends of the Bloom filter size. The cost associated with querying a membership, adding and removing an element in a folded Bloom filter remains the same as with a basic Bloom filter. Central to the notion of folding remains the the foldability of a Bloom filter, which is tightly linked to the folding planing.

One-time folding	Single time unfolding	Sequential folding $\phi_{p_1} \ldots \phi_{p_f}$	Sequential unfolding $\phi_{p_1} \ldots \phi_{p_f}$	Query membership
$o(m)$	$o(m)$	$o(f.m)$	$o(f.m)$	$o(k)$

TABLE II
COST RELATED TO FOLDING AND UNFOLDING

IV. FOLDING PLANNING

Planning the fold (and unfold) of a Bloom filter lies in determining the capacity of the Bloom filter so that this filter offers a great number of possible folds. In order to plan the folding, several factors may be taken into account, including the length of the folding (also called the division factor) and the number of possible successive and identical folds (also called the folding power). In addition, the distribution of the add of items can also exploited. Before moving on with the planning of the Bloom filter capacity, let us first introduce the basic vocabulary and background on numbers that would hereafter help in characterizing the Bloom filter foldability and capacity planning.

A. Preliminaries

Any number $m \in \mathbb{N}^*$ can be uniquely expressed as a product of primes p_j:

$$m = \prod_{j=1}^{\infty}(p_j)^{\gamma_j} = \prod_{j=1}^{a}(p_j)^{\gamma_j} \quad (7)$$

with $p_j \in \mathbb{P}$, and $\forall i < j, p_i < p_j$ with $(i, j) \in \mathbb{N}^2$, $a = max\{i \in \mathbb{N} \ni: \gamma_i > 0\}$.

The number of divisors of m, denoted $d(m)$, is expressed as:

$$d(m) = \prod_{1}^{a}(1 + \gamma_j) \quad (8)$$

$*2$	3	4	$*6$	8	10	$*12$	18	20	24	30	36	48	$*60$
72	84	90	96	108	$*120$ 168	180	240	336	$*360$	420	480	504	540
600	630	660	672	720	840	1080	1260	1440	1680	7560	9240	10080	12600
13860	15120	18480	20160	25200	27720	30240	32760	36960	37800	40320	41580	42840	43680
45360	50400	$*55440$	65520	75600	83160	98280	110880	131040	138600	151200	163800	166320	196560

TABLE III
LARGELY COMPOSITE NUMBERS (EXTRACTED FROM THE ANNOTATIONS PROVIDED AS PART AS [11] FROM THE TABLE HANDWRITTEN BY S. RAMANUJAN). SUPERIOR HIGHLY COMPOSITE NUMBERS ARE UNIDENTIFIED BY *

Any number can be categorized into two disjoints types: prime and composite number (both verify definition 7). A *prime* is a number > 1 that holds no positive divisors other than 1 and itself. A *composite number* has divisor(s) apart from itself and unity. We are interested in composite number and more precisely to the compositiveness of numbers because the higher the compositiveness, the more flexible the folding. In other words, we are concerned by numbers that hold a high number of divisors, which naturally excludes the primes that are characterized by their minimum number of divisors. The so-called highly composite number are herein of special interest. A *highly composite number* [11] is a number that has a larger number of divisors than any number less than itself. More formally, m is highly composite iff

$$\forall m' < m, d(m') < d(m). \tag{9}$$

Unfortunately, these highly composite numbers are very sparse. As illustrated in Table III 6, 11 highly composite numbers are < 831600 ; these numbers are identified by * in the Table. More formally [8], the number of highly composite numbers $\leq x$, denoted $\Psi(x)$, is subject to $\Psi(x) << ln(x)^c$ with c refering to a constant. Thus, highly composite numbers cannot be considered as the only candidate. By relaxing constraints, one may consider a number that is largely composite [11], that is a number that has a greater or equal number of divisors than any number less than itself:

$$\forall m' \leq m, d(m') \leq d(m). \tag{10}$$

Given $\Psi(x)$ the counting function of largely composite numbers, it has been proven that $\exists c$ and $d \ni$: $exp(log^c(x) \leq \Psi(x) \leq exp(log^d(x)$ for any large x. Althought, largely composite number are more numerous than highly composite numbers (see Table III). Moving back to the definition of a number (Eq.7), one may guess that a convenient number is characterized by whether it is composed of little primes - the lower the prime, the greater the number of possible folds - and by the powerful of the primes. Together, they represent both the repetitiveness of the folding as well as the ability to smoothly reduce the size of Bloom filter. Let consider the former case and introduce the *smooth number* which is known as a number with only small prime factors. In particular, a positive integer is said to be y-smooth if it holds no prime factor exceeding y (i.e., any constituing prime factors $\leqslant y$). Smooth numbers are herein useful during the definition of the capacity of a Bloom filter. Indeed, if we choose a number m holding a (very) large[1] prime factor p (i.e., a number close to m), then it remains unlikely that there exist (many) other primes of m. Such a number m should henceforth be avoided to provide a high flexibility in the folding of the Bloom filter. Intuitively, low primes are compensated by high powers and/or a high diversity of primes. In addition, while being fairly numerous, these numbers have a simple multiplicative structure. It is worth mentioning that those y-smooth numbers nearly approach the best candidate for an easy factorization and henceforth for flexible folding. In addition, smooth numbers are dense. Let $\Psi(x, y)$ denote the number of y-smooth integers that are lower than x. Given $y = x^{\frac{1}{u}}$ ($\forall u \geq 1$), this number tends to a non zero limit as $x \to \infty$ giev. The cardinality of $S(x, y)$, denoted $\Psi(x, y)$ verifies:

$$\Psi(x, y) \backsim x.\rho(u) \ \ as\ x \to \infty\ with\ x = y^u \tag{11}$$

$\rho(u)$ is entitled the Dickman de Bruijn and is defined by:

$$\forall u > 1, \rho(u) = \frac{1}{u} \int_{u=-1}^{u} \rho(t)dt \tag{12}$$

The following estimation of ρ is further obtained by differentiating $\rho : \rho(u) = (\frac{e+o(1)}{ulogu})^u$ as $u \to \infty$. As illustrated in Table IV, the proportion of y-smooth integers (and the corresponding bounds) is fair even if $y << x$.

y	$\Psi(x, y)$
$\leq \sqrt{log(x)log(log(x))}$	$\frac{1}{\pi}(y)! \prod_{p \leq y} \frac{log(x)}{log(p)}(1 + o(\frac{y^2}{log(x).log(y)}))$
$= c.log(x)$	$exp([log(1 + c) + c.log(1 + \frac{1}{c})].$ $\frac{log(x)}{log(log(x))}(1 + o(\frac{1}{log(log(x))}))$

TABLE IV
FEW ESTIMATES ON THE NUMBER OF SMOOTH NUMBERS [10]

Synthesis. Highly composite and then largely composite numbers followed by smooth numbers constitute indisputably the best candidates for setting the capacity of a (counting) Bloom filter. The sparsity of the two formers implies the use of smooth numbers that constitute a fair compromise given that the resulting low prime factors are compensated by high powers and/or a high number of primes.

[1]A *rough number* defines a k-rough (or k-jagged) number is a positive integer all of whose prime factors are greater than or equal to k.

B. Generation of Smooth Numbers

Finding the numbers that are y-smooths and that are $\leq x$ can be performed by using the sieve of Eratosthenes. As pointed out in [10], rather than crossing the primes that are $\leq x$ (as it is the case traditionally), one crosses the numbers that are powers of a prime $\leq y$. Then, one counts the number of crosses for each number. If that sum of crosses exceeds a given threshold (as defined hereafter), then the number is defined as a smooth number. We adopt a formulation of the problem of generating y-smooth numbers, which is slightly different. We are interested in finding the y-smooth numbers that pertain to the interval $[x, x+z]$ rather than to the interval $[0, x+z]$.These numbers correspond to the potential candidates that may be selected for setting the (counting) Bloom filter capacity given a range of false positives that is defined by the user. Hereafter, we propose a simple algorithm and its related analytical evaluation.

Finding y-smooth number within [x, x+z] - Let us consider the determination of the smooth numbers that pertain to the interval $[x, x+z] \cap \mathbb{N}$ (see Algorithm 1). The first key step (line 4) consists in generating the set of primes that are $\leq y$. Let $\mathcal{P}_y$ be that set and $\pi(y)$ the number of primes in this set, which given the prime number theorem, is roughly approximated as $\frac{y}{ln(y)}$. The generation of such primes is not resource-consuming given that y is by construction kept low. The number of tests can be reduced by (as with the sieve of Eratosthenes) walking through all the other numbers that are a multiple of that prime. Considering those primes (line 5) and walking through the interval $[x, x+z]$ (line 6), the objective lies in looking for a prime that constitutes a factor of a number (i.e., a prime that divides this number as in line 9). Once identified, this number is crossed (line 12). In order to priviledge addition to the detriment of multiplication, a logarithm formulation is priviledged whenever possible (lines 8, 12, 22). Briefly scheched, this consists in taking logarithmic of numbers. Finally, a smooth number is identified by taking advantage of the following property: a composite number $m = \prod_{j=1}^{a} p_j^{\gamma_j}$ subject to $x \geq m \geq x+z$ verifies:

$$log(x) \leq \sum_{j=1}^{a} \gamma_j log(p_j) \leq log(x+z) \tag{13}$$

As proposed by Pommerance [9] in the context of establishing smoothness tests, an early abort strategies can be applied if the number of prime factors is insufficient at early point. The performance associated with generating such a y-smooth number clearly depends of the number of primes $\leq y$ (i.e., $\pi(y)$) and of the length of the considered interval.

Algorithm 1 Enumerate y-smooth numbers $\in [x, x+z]$

Require: $x \in \mathbb{N}^*, y \in \mathbb{N}^*, z \in \mathbb{N}^*, z > x$

1: **for** $i = 0$ to z **do**
2: w[i] = 0 {initialize the array w used to record the sum of the primes powers}
3: **end for**
4: $\mathcal{P}_y = getPrime(0, y)$ {get the primes > 0 and $\leq y$}
5: **for** $p \in \mathcal{P}_y$ **do**
6: **for** $i = 0$ to z **do**
7: $j \leftarrow 1$
8: **while** $p^j \leq x+z$ $\{j.log(p) \leq log(x+z)\}$ **do**
9: **if** p^j divides $x+i$ **then**
10: $l \leftarrow 0$
11: **while** $i + l.p^j \leq z$ {consider the successive numbers of [x, x+z] that are divisible} **do**
12: $w[i + l.p^j] \leftarrow log(p)$ {cross the number}
13: $l \leftarrow l + 1$
14: **end while**
15: **end if**
16: $j \leftarrow j + 1$
17: **end while**
18: **end for**
19: $i \leftarrow i + 1$
20: **end for**
21: **if** $A[i] \geq log(x)$ **then**
22: $SmoothNumbers = SmoothNumbers \cap x + i$ {$x+i$ is a y-smooth number}
23: **end if**

C. On Adaptive Folding and Unfolding

A (counting) Bloom filter is dynamically resized by folding or unfolding this latter. More precisely, at stage t, this resizing is governed by 3 key factors:

- the number of elements that is stored in the Bloom filter at t, simply denoted n,
- the number of hash functions, denoted k,
- the probability of false positive that is expected.

Let assume a range of false positive, denoted $[\rho - \epsilon, \rho + \epsilon]$ is defined as admissible by the user. Recall that the false positive rate can be expressed as $\rho = (1 - (1 - \frac{1}{m})^{kn})^k \approx (1 - e^{\frac{kn}{m}})$, then the size of the Bloom filter m, at stage t, can be further expressed as:

$$\frac{1}{1 - \sqrt[kn]{1 - \sqrt[k]{\rho + \epsilon}}} \leq m \leq \frac{1}{1 - \sqrt[kn]{1 - \sqrt[k]{\rho - \epsilon}}} \tag{14}$$

Figure 2 illustrates the evolution of the Bloom filter size as a function of the ρ, n with $k = 2$. The problem of determining the next folding can be transposed as finding S_{t+1} with $m_{t+1} = \prod_{j \in S_{t+1}} (p_j)^{\gamma_j}$ subject to $m_{t+1} \in [\frac{1}{1 - \sqrt[kn]{1 - \sqrt[k]{\rho+\epsilon}}}, \frac{1}{1 - \sqrt[kn]{1 - \sqrt[k]{\rho-\epsilon}}}]$. Toward this goal, several policies can be proposed.

The most naive form consists in taking a folding/unfolding decision regardless of the previous folds. Assuming that the capacity of the Bloom filter were initially set to $m_1 = \prod_{j \in S} p_j^{\gamma_j}$, the folding that should be performed, if necessary[2], is charac-

[2]If m is about to violate eq. 14

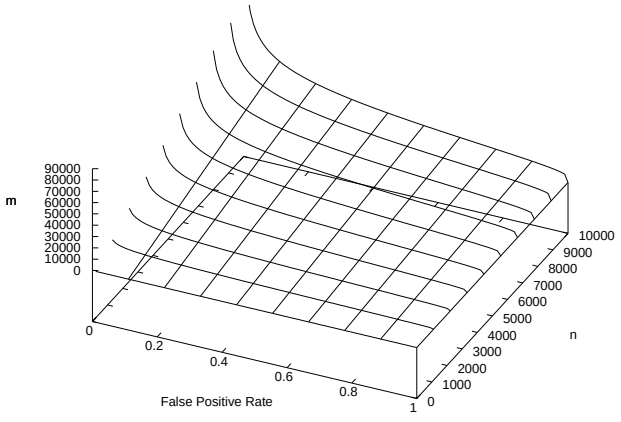

Fig. 2. Bloom Filter Sizing (with k=2)

terized by a reduction factor corresponding to $\prod_{j \in S'} (p_j)^{\gamma_j}$, with $S' \subset S$.The cost associated with determining S' is bounded by the number of possible combinaison of divisions of m_1 (i.e., $\prod_{j=1}^{a} (\gamma_j + 1) - 1$) while the cost related to the folding is $o(m_{t+1})$. Note that a cost-efficient optimisation lies in taking advantage of the previous foldings, potentially following e.g., a greedy algorithm so as to keep to a minimum the cost related to exploring possible folding options. Alternativelly, an opposite optimisation lies in using the entropy as the metric rather than the computation cost. More preciselly, as pointed by Shanon, changing the folding (and more precisely, considering folding that have not been yet performed) increases the entropy, and by consequence the amount of information carried while reducing the probability of false positive.

V. Conclusion

The Internet of the things has reached a stage that enables an easy access to information and services anywhere, anytime. However, such vision still comes with practical limitation mainly relating to limited bandwidth and energy. It is henceforth crucial to devise novel solutions for supporting lightweight networking, data flow and service access so as to impact as less as possible bandwidth and energy. This paper touches upon such an issue by resizing the Bloom filter, which hence permits to keep to a minimum the bandwith and energy usage associated with exchanging a Bloom filter. The basic idea consists in folding or unfolding a Bloom filter so that the false positive rate keeps neglicted. We acknowledge that the halve of a Bloom filter were originally suggested in [13] and applied [4] so as to reduce the bandwidth consumption induced by the exchange of Bloom filters in the context of large-scale Grid [14]. We herein generalize this approach by introducing the concept of foldin/unfolding along with a novel formulation of the problem: the key challenge consists in determining how a folding should be performed, namely the number of times that Bloom filter should be folded/unfolded and the reduction factor associated with each fold/ unfold. We formulate that as an off-line planing of the factorization of an integer (corresponding to the Bloom filter size) and further proposed directions for optimising the dynamic folding/unfolding of a Bloom filter.

Acknowledgment

Authors would like to thanks the creators of the game paperPlane for its so inspiring nature: http://www.youtube.com/watch?v=jof_1ByCtKM, paperplane-game.com. This work has been supported through the visitor fellowship provided by SRI and by the french national agency (ANR) under contract ANRBLAN-SIMI10-LS-100618-6-01.

References

[1] P. Almeida, C. Baquero, N. Preguica, and D. Hutchison. Scalable bloom filters. *Information Processing Letters*, 101, 2007.

[2] H.B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communication with ACM*, 13(7), 1970.

[3] L. Fan, P. Cao, J. Ameida, and A.Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM transac. on netzorking*, 8, 2000.

[4] A. Gehani, B. Baig, S. Mahmood, D. Tariq, and F. Zaffar. Fine-grained tracking of grid infections. In *ACM IEEE International Conference on Grid Computing (GRID)*, 2010.

[5] D.K Guo, H.H Chen, J. Wu, and X.S Luo. Theory and network application of dynamic bloom filters. *IEEE Global Telecommunicatiion conference*, 2006.

[6] Z. Heszerger, J. Tapolcai, A. Guyas, and al. Adaptive bloom filters for multicast addressing. *High-Speed Netzorks workshop*, 2011.

[7] M. Mitzenmacher. Compressed bloom filters. *IEEE ACM Transaction on networking*, 10, 2002.

[8] J.-L. Nicholas. On highly composite numbers. In *In Ramanujan Revisited: Proceedings of the Centenary Conference, University of Illinois at Urbana-Champaign, Ed. G. E. Andrews, B. C. Berndt, and R. A. Rankin). Boston, MA: Academic Press*, pages 215–244, 1998.

[9] C. Pomerance. Analysis and comparison of some integer factoring algorithms. In *Computational methods in number theory*, pages 89–139, 1982.

[10] C. Pomerance. The role of smooth numbers in number theoretic algorithms. In *International Congress of Mathematicians*, pages 411–422, 1994.

[11] S. Ramanujan. Highly composite numbers,. In *The Ramanujan Journal annoted by J.L Nicolas and G. Robin, Kluwer Academic publishers*, pages 119–153, 1997.

[12] F. Sailhan and V. Issarny. Scalable service discovery for manet. In *IEEE International Conference on Pervasive Computing and Communications (PERCOM)*, 2005.

[13] D.B. Siano and J.D. Siano. An algorithm for generating highly composite numbers. In *Internet Mathematics*, pages 636–646, 1994.

[14] D. Tariq, B. Baig, and A. Gehani. Identifying the provenance of correlated anomalies. In *26th ACM Symposium on Applied Computing (SAC)*, 2011.

[15] K. Xie, Y. Min, and D. Zhang. A scalable bloom filter for membership queries. *IEEE Global Telecommunicatiion conference*, 2007.