Folding and Unfolding Bloom Filters - an Off-line Planing and on Line-Optimisation

Francoise Sailhan, Mark-Oliver Stehr

Folding and Unfolding Bloom Filters - An Off-Line Planning and On-Line Optimization Problem

Francoise Sailhan*†, and Mark-Oliver Stehr*
* Computer Science Laboratory, SRI international
Menlo Park, CA 94025 USA
†Cedric Laboratory, CNAM, 75003 Paris, France
Emails: francoise.sailhan@cnam.fr, mark-oliver.stehr@sri.com

***Abstract*—The Internet of things has reached a stage that allows ubiquitous data access. Still, practical limitations remain in networks with scarce bandwidth. Here, we examine the Bloom filter data structure and its use in distributed protocols. We discuss how to minimize the bandwidth and energy usage consumed when distributed protocols exchange Bloom filters, through dynamic Bloom filter resizing. We propose a general and novel formalization of Bloom filter resizing, through foldings and unfoldings. The key challenge in the folding approach is determining suitable parameters and how to perform a folding. Specifically, we address the number of times that a Bloom filter should be folded and optionally unfolded, and how to determine an ideal reduction factor for this process. We formulate our approach as off-line planning of the integer factorization problem (where the integers correspond to the size of a Bloom filter), and propose further directions for optimizing the dynamic folding and unfolding of a Bloom filter.**

I. Introduction

The last decade is characterised by the convergence of pervasive technologies including wireless communication, smart devices and the Internet. The resulting Internet of things enables not only users but also things to access resources anywhere, anytime. In practice, data flows ubiquitously using global wireless connectivity (*e.g.*, 3G) or *ad-hoc* local area networking, as enabled by, for example, IEEE 802.15.4 protocols. Two key limitations of these networks remain the limits on bandwidth and energy. In this context, early-stage data and resource discovery should be provided so as to allow efficient data dissemination [14]. A Bloom filter [2] can play a crucial role by allowing to represent a set of data in a compact manner and by supporting efficient membership queries. More precisely, a Bloom filter is a vector of bits. It is distinguished by the fact that the time associated with a membership query 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 to be added to the Bloom filter is first hashed. Typically, $k > 1$ hash functions are used. Then, the k outputs are used to flip to 1 the bits at the related positions in the Bloom filter. In order to check whether an element is stored in the Bloom filter, the queried element is hashed (relying on the same hashing functions). If the k bits at the related position are set to 1 in the Bloom filter, this element is stored. It is worth mentioning that false negatives do not occur. Nevertheless, the approximate nature of Bloom filter implies that false positives might appear. The probability of a false positives depends on (i) the Bloom filter size, (ii) the number of hash functions and (iii) the number of elements stored. Keeping the false positive rate to a minimum requires dynamic adaptation of the number of hash functions and/or of the Bloom filter size. Further attempts to increase the size of the Bloom filter have appeared in the literature. The basic idea is to create a set of Bloom filters [6], [16], [1]. As a side effect, a membership query is a function on the size of this set. While these methods deal with the increase of the Bloom filter size, the related decrease is neglected. An alternative [7] consists in partitioning the Bloom filter into k disjoint sub-ranges wherein an element is added by hashing the sub-range with the related hash function h_k. Depending on the expected probability of a false positives, k is either dynamically increased or decreased, hence leading to a reduced or increased Bloom filter size. Nevertheless, changing the number of hash functions offers lower granularity than resizing the Bloom filter.

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 reduce/increase the size of the Bloom filter by folding/unfolding it so that the false positive rate ρ stays in the following interval $[\rho - \epsilon, \rho + \epsilon]$ and the increase of memory size is kept to a minimum. We acknowledge that halving a Bloom filter was originally suggested in [3] and successfully applied [5] 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 computing [15]. We herein extend and generalize this approach by introducing the concept of folded and unfolded Bloom filters, which permits highly flexible resizing.

We also introduce a novel formulation of the problem, which allows to keep the false positive rate in check by injecting entropy into the folding process. 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 folding/unfolding. We show that this can be formulated as (i) an off-line planning problem of the factorization of an integer (namely the Bloom filter size) and (ii) an on-line optimization problem of the Bloom filter folding/unfolding. Our main

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

contribution is twofold: we propose an algorithm for planning the folding and unfolding 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 planning. In addition, based on these planned foldings/unfoldings, we propose an approach to dynamically optimizing the folding and unfolding of a Bloom filter.

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

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.*, $\forall i \in [0, m]$, $b(i) = 0$. The vector is then updated by adding additional elements.

a) Bloom filter update: In order to add an element e to 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.

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

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 any 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, a few proposals have emerged. The so-called *counting Bloom filter* [4] is the most popular. A counting vector $c_1, \cdots, c_m$ replaces the bit vector of the Bloom filter. It instead 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 $h_1(e), \cdots, h_k(e)$. The size of the counting vector is greater than the Bloom filter size by a factor that depends on the space allocated to count. These counters must be selected to be large enough 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 required to insert or find an element depends on the number of hash functions and is hence $o(k)$.

2) Both basic and counting Bloom filters engender false positives. Given n the number of elements that are stored in the [counting] Bloom filter (see Table I) and assuming that the hash functions are perfect, the probability of a false positive, denoted ρ, satisfies:
$$\rho = (1 - (1 - \frac{1}{m})^{kn})^k \approx (1 - e^{\frac{kn}{m}}) \qquad (1)$$
Unlike a basic Bloom filter, a counting Bloom filter may produce a false negative, which occurs if the counting vector is undersized and generates 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, and
   - adding the vectors of 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 a 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 subtraction.

Despite the above promising properties, [counting] Bloom filters suffer 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 and/or an increased false positive rate. A 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 for increasing the Bloom filter size are slightly different [6], [16], [1]. All consist in using a set of Bloom filters, that is, 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 [6], the Bloom filters are of equal size: a matrix $m \times s$ is handled, with $s = \lceil \frac{n}{n_0} \rceil$ where n is the actual number of items added to the matrix and n_0 denotes 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 a basic Bloom filter (namely $o(k)$). Nevertheless, the time associated with seeking an element increases.Similar approaches are proposed in [16], [1], except that the size of the added Bloom filters grows exponentially in the former (*i.e.*, if $n > 2^{i-1}n_0$ then $m = i \times m_0$), and Bloom filters are added so that the probability of false positives grows geometrically $p_o, p_0.r, p_o r^2$. Together, these approaches lead to an increase of the lookup time that is characterised by a factor s. In addition, these approaches focus on increasing the Bloom filter size, but the decrease is not addressed. Such a decrease could be accomplished through compression [9]. It is worth mentioning that counting Bloom filters could not be easily applied here. An alternative to increasing the [counting] Bloom filter size lies in changing the number of hash functions [7]. 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 makes it possible to easily increase or decrease the [counting] Bloom filter size, as needed. In addition, adding an item can be parallelised and the asymptotic performance of a partitioned Bloom filter remains the same as that of a Bloom filter. However, performance of a partitioned Bloom filter is worse due to the use of disjoint ranges.

In order to tackle this issue, we propose adapting the Bloom filter size by dynamically folding and unfolding it. As pointed out in [3], and promoted and successfully applied in [5], a nice feature of Bloom filters is that they can be halved in size, assuming 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 needed to transmit a Bloom filter [5].

Going one step further, we generalize this approach by:

- introducing the notion of folding and unfolding of a [counting] Bloom filter, which allows increasing or decreasing the Bloom filter size in a flexible manner;
- providing a formulation of the problem of planning the foldings and unfoldings, while proposing an analytical and experimental evaluation; and
- formally describing the process of folding/unfolding and proposing further approaches for optimizing these operations.

III. Dynamic Bloom filters

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

Fig. 1. Folding

- We are concerned with foldings in one dimension,
- The reduced size of the Bloom filter remains an integer. This is necessary so that a reduced Bloom filter remains a Bloom filter (*viz.*, an array of bits, whose size is an integer) after the reduction,
- The foldings are of equal size. This is necessary to enforce an equi-probable repartition of the bits in the resulting [folded, counting] Bloom filter.
- 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 no impact on the resulting folded Bloom filter.
- A folding is characterized by:
  - a *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); that is, a Bloom filter of 50 elements is folded into a Bloom filter of 10 elements; and
  - a *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 two times. The rank of the reduction factor 5 is hence 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: $50 = 5^2 \cdot 2$. Note also that 5 and 2 are primes.

The same folding applies to a counting Bloom filter. Unless especially pointed out, the term Bloom filter will hereafter encompass the notions of basic and counting Bloom filters together.

A. Folding Planning

Let formally define 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 powers 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 ensure that the Bloom filter is highly foldable, the number m_1 should be selected so that (i) it corresponds to a composite number (rather than a prime) and (ii) there exist 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 are three ways of folding a Bloom filter:

- An *elementary folding* is a single folding characterised by a small 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* is a single folding characterised by a factor of reduction corresponding to a composite (non-prime) number. For instance, a Bloom filter size can be reduced by a factor of 4.
- A *sequential folding* is a succession of elementary and/or composite foldings. In Figure 1, a sequence of elementary foldings is furnished.

Elementary Folding - We consider the elementary folding of a Bloom filter B_t, which results in a folded Bloom filter denoted B_{t+1}. Let p_l be the reduction factor of this initial folding and ϕ_{p_l} represent the folding function. This reduction factor can be expressed as the 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, namely,

- a basic Bloom filter is folded using the logical OR operator. More precisely, this consists in applying a bit-wise OR on the Bloom filter portions that are folded, and
- a counting Bloom filter is folded by adding vectors corresponding to the different parts of the counting Bloom filter that is folded.

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

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

Thus, the resulting Bloom filter can be expressed as:

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

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

The computing cost associated with this elementary folding is $O(m_t)$, that is, it depends of the size of B_t. The probability of false positives ρ on the folded Bloom filter B_{t+1} containing n elements, using k hashing functions satisfies:

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

In other words, the probability of false positives of a folded Bloom filter is the same as that of a Bloom filter with the same size. Overall, such an elementary folding can be composed and/or sequentially performed so as to further reduce the Bloom filter size.

Composite and Sequential Folding - Composite and sequential folds behave differently. Whereas the composite folding is performed once with a high reduction factor, a sequential folding is done by successively applying either elementary and/or composite foldings. Despite this difference, the Bloom filters resulting from a composite folding ($\phi_{p_r.p_l}$) and sequential folding (one time folding $\phi_{p_r}.\phi_{p_l}$) are identical.

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

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

Proof: Given a Bloom filter B_t, let prove 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:

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

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

Thus, $\forall i \in [1, m_{t+2}], b_{t+2}(i)$ can be expressed as:

$$\begin{aligned} b_{t+2}(i) &= \oplus_{l=0}^{p_l-1} \oplus_{r=0}^{p_r-1} b_t(i + l \cdot m_{t+2} + r \cdot m_{t+1}) \\ &= \oplus_{l=0}^{p_l-1} \oplus_{r=0}^{p_r-1} b_t(i + l \cdot \frac{m_t}{p_l.p_r} + r \cdot \frac{m_t}{p_r}) \end{aligned}$$

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

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

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, $j = l + r p_l$ $\square$.

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 of a sequential folding is greater than the cost of a composite folding. Specifically, a sequential folding is $O\Big(m_t \cdot (1 + \frac{1}{p_l})\Big)$, and a composite folding $\phi_{p_r.p_l}$ is $O(m_t)$.

Theorem - The sequential folding, denoted o, of two 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 matter. 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 fold; the folding pathway cannot be inspected given the resulting Bloom filter.

Until now, we have focused our study on the folding. Now we analyze the reverse operation.

Unfold. With Bloom filters, the unfolding cannot be expressed as a simple operation, *e.g.*, an AND or a subtraction. Let B_t represent a Bloom filter (at Stage t) that is folded resulting in B_{t+1}. Assume 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:

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

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

Membership Query - Querying whether an item q belongs to a Bloom filter is straightforward. Assume 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, the element q is not stored. Note that, given that the Bloom filter has been folded, the position of the investigated bit will be divided by the same factor. In practice, such a division consists in taking the modulo, denoted mod, of the hashed item $h_k(q)$. Thus, the bits that are inspected are those that are located at position $mod\big(\cdots mod\big(mod\big(h_k(q), p_{j_1}\big), p_{j_2}\big), \cdots, p_{j_f}\big)$. More formally, q is not stored iff:

$$\exists k \text{ so that } B\left[mod\big(h_k(q), \prod_{i=0}^{f}(p_{j_i})\right] = 0 \qquad (6)$$

The cost associated with a membership query 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.

Synthesis - A folded Bloom filter results from a sequential composition of elementary and/or composite foldings. 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 accrues an additional cost that depends on the Bloom filter size. The cost associated with querying membership of, adding, or 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 degree of freedom when planning of the folding.

Single query	Single unfolding	Sequential folding $\phi_{p_1}...\phi_{p_f}$	Sequential unfolding $\phi_{p_1}...\phi_{p_f}$	Membership folding
$O(m)$	$O(m)$	$O(f \cdot m)$	$O(f \cdot m)$	$O(k)$

TABLE II
COST RELATED TO FOLDING AND UNFOLDING

IV. FOLDING PLANNING

Planning the folding (and unfolding) of a Bloom filter lies in determining the capacity of the Bloom filter so that this filter offers a large number of possible folds. In order to plan the folding, several factors may be taken into account, including the factor of the folding (also called the division factor) and the number of possible successive identical foldings (also called the folding power). In addition, the rate of addition of items can also exploited. Before moving on with the planning of the Bloom filter capacity, let first introduce the basic vocabulary and background on the number theory that will subsequently help.

A. Preliminaries

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

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

with $p_j \in \mathbb{P}$, and $\forall i < j, p_i < p_j$ with $(i,j) \in \mathbb{N}^2$, $a = max\{i \in \mathbb{N}/\gamma_i > 0\}$.
The number of divisors of m, denoted $d(m)$, is expressed as:

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

Any number can be categorized as one of two disjoint types: a prime or composite number. Both satisfy 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 one. We are interested in composite numbers and more precisely, in the compositeness of numbers because the higher the compositeness, the more flexible the folding. In other words, we are concerned with numbers that have a

∗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 [13] FROM THE TABLE HANDWRITTEN BY S. RAMANUJAN). SUPERIOR HIGHLY COMPOSITE NUMBERS ARE IDENTIFIED BY *

high number of divisors, which naturally excludes the primes, characterized by their minimum number of divisors. The so-called *highly composite numbers* are herein of special interest. A highly composite number [13] 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 7, eleven highly composite numbers are < 831600 ; these numbers are identified by * in the Table. More formally [10], the number of highly composite numbers $\leq x$, denoted $\Psi(x)$, is subject to $\Psi(x) << ln(x)^c$ with c a constant. Thus, highly composite numbers cannot be considered the only candidates. By relaxing constraints, one may consider a number that is largely composite [13], 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 so that $exp(log^c(x)) \leq \Psi(x) \leq exp(log^d(x))$ for any large x. Is is noteworthy that largely composite numbers are more numerous than highly composite numbers (see Table III). Moving back to the prime decomposition 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 powers of the primes. Together, they represent both the repetitions of folding as well as the ability to smoothly reduce the size of Bloom filter. We consider the former case and introduce the *smooth number,* which is known as a number with only small prime factors [8]. In particular, a positive integer is said to be y-smooth if it holds no prime factor exceeding y *(i.e.*, any constituting prime factors $\leqslant y$). Smooth numbers are useful for defining 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 therefore be avoided to assure a high flexibility in the folding of the Bloom filter. Intuitively, low primes are compensated for by their high powers and/or by a high diversity of primes. In addition, while being fairly numerous, these numbers have a simple multiplicative structure.

We believe that y-smooth numbers are the best candidates 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 lower than x. Given $y = x^{\frac{1}{u}}$ ($\forall u \geq 1$), this number tends to a non-zero limit as $x \to \infty$. The cardinality of $S(x, y)$, denoted $\Psi(x, y)$, satisfies:

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

$\rho(u)$ is entitled the Dickman de Bruijn function 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 $u \to \infty$.

Synthesis. Highly composite and then largely composite numbers followed by smooth numbers constitute the best candidates for setting the capacity of a [counting] Bloom filter. The sparsity of both highly composite and largely composite numbers suggests the use of smooth numbers, which constitute a fair compromise given that the resulting low prime factors are compensated for by high powers and/or a high number of primes.

B. Generation of Smooth Numbers

The numbers that are y-smooth and that are $\leq x$ can be found using the sieve of Eratosthenes. As pointed out in [12], rather than crossing out the primes that are $\leq x$ (as it is done traditionally), one crosses out the numbers that are powers of a prime $\leq y$. Then, one counts the number of crossed-out powers for each number. If that count exceeds a given threshold (defined hereunder), then the number is 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 capacity of the initial (*i.e., unfolded*) [counting] Bloom filter given a range of false positives that is defined by the user. We propose a simple algorithm and its related analytical evaluation.

Finding *y*-smooth Numbers within [x, x+z] - Let consider the determination of the smooth numbers that pertain to the interval $[x, x + z] \cap \mathbb{N}$ (see Algorithm IV-B).

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

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-intensive given that y is by construction kept low. The number of tests can be reduced (as with the sieve of Eratosthenes) by walking through all the other numbers that are multiples of that prime. Considering those primes (line 5) and walking through the interval $[x, x+z]$ (line 6), the objective is to find a prime that is a factor of a number (*i.e.*, a prime that divides this number as in line 9). Once identified, this number is crossed out (line 12). In order to privilege addition over multiplication, a logarithm formulation is privileged whenever possible (lines 8, 12, 22). Briefly sketched, this consists of computing the logarithm. 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 $x \geq m \geq x+z$ satisfies:

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

As proposed by Pommerance [11] in the context of establishing smoothness tests, an early-abort strategy can be applied if the number of prime factors is insufficient at early stage. The performance associated with generating such a y-smooth number clearly depends on the number of primes $\leq y$ (*i.e.*, $\pi(y)$) and of the length of the considered interval.

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** $p^j \leq x + z$ $\{j \cdot log(p) \leq log(x+z)\}$ **do**
9: **if** p^j divides $x + i$ **then**
10: $l \leftarrow 0$
11: **while** $i + l \cdot p^j \leq z$ {consider the successive numbers of [x, x+z] that are divisible} **do**
12: $w[i + l \cdot p^j] \leftarrow log(p)$ {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 it. More precisely, at Stage t, this resizing is governed by three 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; and
- the expected probability of a false positives.

Assume a range of false positives, denoted $[\rho - \epsilon, \rho + \epsilon]$ is defined as admissible by the user. Recall that the false positives 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 high flexibility in the folding that can be exploited by applications and is witnessed by the large number of suitable smooth numbers and hence reduction factors to choose from for a given fixed interval (vertical axis) selected by the user. The problem of determining the next folding can

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

be formulated 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 used. The simplest policy consists of taking a folding/unfolding decision regardless of the previous steps. Assuming that the capacity of the Bloom filter was initially set to $m_1 = \prod_{j \in S} p_j^{\gamma_j}$, the folding that should be performed, if necessary[2], is characterized by a reduction factor corresponding to $\prod_{j \in S'} (p_j)^{\gamma_j}$, with $S' \subset S$. The cost of determining S' is bound by the number of possible combinations 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, for example, a greedy algorithm so as to minimize the cost of exploring possible folding options. Another optimization lies in using entropy in the choice of the folding as the metric rather than the computation cost. More precisely, changing the folding with each transmission (and considering foldings that have not been yet performed) increases the entropy and by consequence the amount of information carried, which in turn leads to a reduction in the probability of false positives if we aggregate multiple transmissions.

[2]If m does not satisfy Eq. 14

V. Conclusion

The Internet of things has reached a stage that enables easy access to information and services anywhere, anytime. However, such vision still comes with practical limitations mainly relating to bandwidth and energy constraints that are especially difficult to overcome on mobile devices. It is therefore crucial to devise novel solutions for supporting lightweight networking, data flow and service access. This paper explores a new direction by resizing the Bloom filter, which hence minimizes the bandwith and energy usage associated with exchanging a Bloom filter. The basic idea consists of selecting suitable Bloom filter parameters and folding or unfolding a Bloom filter so that the false positive rate satisfies the application needs. We acknowledge that halving a Bloom filter was originally suggested in [3] and applied [5] to large-scale grid computing [15]. We herein generalize this approach by introducing the concept of folding/unfolding along with a novel formulation of the problem: The key challenge consists in determining the right parameters for the initial Bloomfilter and how a folding should be performed, in particular the number of times the Bloom filter should be folded/unfolded and the reduction factor associated with each folding/unfolding. We have formulated that as an off-line planning problem of the factorization of an integer (corresponding to the Bloom filter reduction factor) and further proposed directions for optimizing the dynamic folding/unfolding of a Bloom filter.

Acknowledgments

The authors would like to thank the creators of the game paperPlane for inspiration (http://www.youtube.com/watch?v=jof_1ByCtKM, paperplane-game.com). The authors appreciate thoughts and feedback from Ashish Gehani and Sam Wood, especially for many future research directions. Partial support from National Science Foundation Grant 0932397 (A Logical Framework for Self-Optimizing Networked Cyber-Physical Systems) is gratefully acknowledged. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NSF. This work has been also partially supported 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] A. Broder and M. Mitzenmacher. Network applications of Bloom filters: A survey. In *Internet Mathematics*, pages 636–646, 2002.

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

[5] 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*, 2010.

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

[7] Z. Heszerger, J. Tapolcai, A. Guyas, and al. Adaptive Bloom filters for multicast addressing. *High-Speed Networks workshop*, 2011.

[8] J. M. Reyneri M. E. Hellman. Fast computation of discrete logarithms in GF(q). In *Advances in Cryptology: Proceedings of CRYPTO '82*, pages 3–13, 1983.

[9] M. Mitzenmacher. Compressed Bloom filters. *IEEE/ACM Transaction on networking*, 10, 2002.

[10] J.-L. Nicholas. On highly composite numbers. 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.

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

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

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

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

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

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