A Power Management Proxy with a New Best-of-N Bloom Filter Design to Reduce False Positives

Miguel Jimeno and Ken Christensen
Computer Science and Engineering
University of South Florida
Tampa, FL 33620
{mjimeno, christen}@cse.usf.edu

Allen Roginsky
Computer Security Division
National Institute of Standards and Technology
Gaithersburg, MD 20899
allen.roginsky@nist.gov

Abstract

Bloom filters are a probabilistic data structure used to evaluate set membership. A group of hash functions are used to map elements into a Bloom filter and to test elements for membership. In this paper, we propose using multiple groups of hash functions and selecting the group that generates the Bloom filter instance with the smallest number of bits set to 1. We evaluate the performance of this new Best-of-N method using order statistics and an actual implementation. Our analysis shows that significant reduction in the probability of a false positive can be achieved. We also propose and evaluate a new method that uses a Random Number Generator (RNG) to generate multiple hashes from one initial "seed" hash. This RNG method (motivated by a method from Kirsch and Mitzenmacher) makes the computational expense of the Best-of-N method very modest. The target application is a power management proxy for P2P applications executing in a resource-constrained "SmartNIC".

Keywords: *Networks, Power management, Bloom filter*

1. Introduction

The Internet and the devices that connect to it are consuming an increasing amount of electricity. It is estimated that the Internet is consuming 2% of all the electricity consumed in the US [11]. Electronic devices – most of them connected to the Internet – are the fastest growing consumer of electricity. Reducing the electricity used by desktop PCs and other devices connected to the Internet is thus of growing importance. An average PC consumes 120 W when fully powered-on [20]. Such a PC if fully powered-on 24 hours per day, 365 days per year would add about 10% to the typical US residential electricity consumption (see the appendix for the calculation). This is a non-trivial impact. Many new and emerging network applications are driving PC on-time to increase. One such application is peer-to-peer (P2P) file sharing. A PC running a P2P application is nearly idle 99% of the time (see the appendix for the calculation), but must remain "on the net" so that other P2P users can query the PC to learn if a requested file is being shared.

We are investigating how a P2P application can be executed in a small, low-power microcontroller that proxies for a PC. The PC could then be power managed (that is, enter a low-power sleep state) during the time that shared files are not actively being uploaded or downloaded. The design of a power management proxy is outlined in this paper. The proxy microcontroller is limited in memory and processing capability. It cannot easily contain a large list of file names (strings) and could require significant time to search a large list for a given file name. Bloom filters [1] are an ideal space-efficient, probabilistic data structure for representing a list of file name strings such that testing for membership in the list of files can be done quickly. A Bloom filter is an array of bits. A string is added to a Bloom filter by inputting it to a group of k hash functions resulting in k array index values where each indexed array position is set to 1. A string is tested for membership by inputting it to the same group of k hash functions. If all k generated array positions are set to 1, then the string is probably a member. Non-member strings may map to set bit positions in the Bloom filter array, thus false positives are possible. The probability of false positive of a Bloom filter is increasing with the number of bits set to 1 (we give a precise relationship later in the paper). We investigate using N groups of hash functions to generate N Bloom filters and selecting the group of hash functions that generates the Bloom filter instance with the smallest number of set bits (and thus also the smallest probability of false positive). The selected group of *Best-of-N* hash functions is then used to test queried file names for membership in the list of shared files.

This material is based on work supported by the National Science Foundation under Grant No. 0520081.

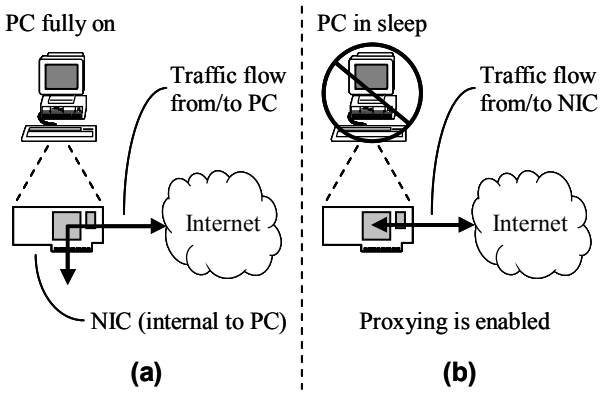

Figure 1. The SmartNIC with proxy capability

The remainder of this paper is organized as follows. In Section 2 we review Bloom filters and describe our P2P proxy in the context of a SmartNIC. Section 3 formally describes our *Best-of-N* method. Section 4 describes our new method for generating multiple hash values from one actual hash. Section 5 analyzes the *Best-of-N* method using order statistics. Section 6 is an experimental evaluation. Section 7 describes related work. Finally, Section 8 is a summary and also describes future work.

2. Using a Bloom Filter in a SmartNIC

PCs connect to the Internet typically via an Ethernet Network Interface Controller (NIC). The NIC may be a chip on the motherboard, or a separate add-on adapter card. The NIC supports the Ethernet PHY and MAC protocols and cannot generate or respond to higher-layer packets. It is the TCP/IP implementation within the PC operating system and the applications running in the PC that generate and respond to packets. A PC needs to be fully powered-on (i.e., processor running) in order to generate and respond to packets. If the PC is powered-down (e.g., to a power management sleep state), then applications executing in the PC are dormant. Any application with TCP connections will lose its connections when a PC goes to sleep. A sleeping PC cannot generate or respond to packets (with the exception of being able to wake-up for specifically defined "wake-up" packets).

Network applications, such as P2P file sharing, are driving up PC on time. These applications require a PC to be fully powered-on at all times to maintain TCP connections and respond to messages. For the majority of the time, PCs running P2P applications are not actively transferring files, but are only receiving and responding to query messages. In order to enable PCs running P2P applications to be power managed, we are studying the idea of using a small low-power controller to "cover", or proxy, for a sleeping PC. This proxy controller will be able to maintain P2P TCP connections and respond to query messages. When an HTTP GET request for a file download is received, the sleeping system is woken-up and control transfers from the proxy controller to the PC. We are exploring locating the proxy controller directly on the NIC, thus it is a "SmartNIC". Figure 1 shows the SmartNIC. In Figure 1(a) the SmartNIC is operating as a standard NIC passing all packets to and from the fully powered-on PC. When the PC goes to sleep, shown in Figure 1(b), the SmartNIC enables proxying and responds to all packets and wakes-up the PC only when its full resources are needed (e.g., for a file transfer).

In order to keep SmartNIC costs low, the proxy controller is limited in both computational and memory capabilities. To maintain a large list of file names being shared – the case for a P2P application – would require considerable memory. A Bloom filter is ideally suited to "compressing" a list of filenames or keywords. We thus study the application of Bloom filters to proxying for a P2P application and how to make a Bloom filter as efficient as possible. Two key measures of performance for a Bloom filter are 1) the probability of false positive and 2) the computational effort required to test for membership. The computational effort required for generating a Bloom filter is also a performance measure. However, in our case the application in the PC will generate the Bloom filter (to be passed to the proxy controller). The PC contains a very powerful processor compared to the proxy controller. Thus, we seek a trade-off in computation whereby generating a Bloom filter can take greater computational effort in order to reduce its probability of false positive. We also seek to minimize computation required to test for element membership.

3. The Best-of-N Method

The *Best-of-N* method uses additional computation in generating a Bloom filter to reduce the probability of false positive. For a given list of elements (e.g., file name strings) to be mapped into a Bloom filter, the *Best-of-N* method finds a Bloom filter instance with the least numbers of bits set to 1. Using N different groups of hash functions, N instances of a Bloom filter are generated sequentially. The instance with the least number of bits set to 1 defines the Bloom filter instance and the hash group that is then used to check for membership in the filter. The *Best-of-N* method is shown in Figure 3 (Figure 2 defines the variables used). Two functions are given, one to generate the Bloom filter and the other to test for membership. A Bloom filter has m bits. The constant K is the number of hash functions in a hash group (corresponds to k hash functions).

bloom[ ]	Bloom filter of length *m* bits
tempBloom[ ]	Temporary Bloom filter
strList	Input list of strings
nextString	Next string in input list of strings
hashValue	Hash value (value from 1 to *m)*
setCount	Current count of bits set to 1
minCount	Minimum count of bits set to 1
hashGroup	Index of best hash group
inString	Input string to test for membership
memberFlag	Membership flag
i, j	Loop counters

Figure 2. Variables for *Best-of-N* method

```
function mapBloom(strList)
clear bloom[ ]
minCount = m

for i = 1 to N do
  open strList
  clear tempBloom[ ]
  setCount = 0
  while (strings remain in strList) do
    nextString = next string in strList

    for j = 1 to K do
      hashValue = hash(nextString, (i–1)*K+j)
      if (bloom[hashValue] == 0)
        increment setCount
        bloom[hashValue] = 1
        if (setCount > minCount)
          continue to outside for loop

    if (setCount < minCount)
      minCount = SetCount
      bloom = tempBloom
      hashGroup = i

return(bloom[ ], hashGroup)

function testBloom(inString, hashGroup)
memberFlag = true
for i = 1 to K do
  hashValue =
  hash(inString, (hashGroup–1)*K+i)
  if (bloom[hashValue] == 0)
    memberFlag = false
    break
return(memberFlag)
```

Figure 3. *Best-of-N* method for Bloom filter

```
function mapString(nextString)
seedValue = hash(nextString)
bloom[seedValue] = 1
seedRNG(seedValue)
for i = 1 to K do
  hashValue = randInt();
  bloom[hashValue] = 1;

function testString(inString)
memberFlag = true
seedValue = hash(inString)
if (bloom[seedValue] == 0)
  memberFlag = false
if (memberFlag == true)
  seedRNG(seedValue)
  for i = 1 to K do
    hashValue = randInt();
    if (bloom[hashValue] == 0)
      memberFlag = false
      break
return(memberFlag)
```

Figure 4. RNG hash method for Bloom filter

In Figure 3, the function hash(string, i) generates a hash value for string using hash function i. One way to implement multiple hash functions from a single hashing method is to seed the method with a value i. For example, we use CRC32 as a hashing method and seed the CRC accumulator with different values to obtain different hash functions.

Intuitively, the larger the N value the lower the probability of false positive and the longer the time required to compute the Bloom filter. In Section 5 we analyze the reduction of probability of false positive as a function of N.

4. Generating Hash Values with an RNG

Given two hash functions, $h_1(x)$ and $h_2(x)$, additional pseudo hash values can be generated as

$$g_i(x) = h_1(x) + i \cdot h_2(x) \qquad (1)$$

where i is the hash value index and x is the string value being hashed. Kirsch and Mitzenmacher [12] describe the application of this method to Bloom filters. To generate k indexes into a Bloom filter requires only two actual hashes and $k-2$ iterations of (1) which is a considerable reduction in processing compared to needing k actual hashes. It is shown in [12] that using this method does not increase the asymptotic probability of false positive.

We build upon this by using a linear congruential generator (LCG) random number generator (RNG) where a single hash value is used as a seed to generate additional pseudo hash values. Figure 4 shows the method for mapping strings into, and testing for membership in, a Bloom filter. The value seedValue is the initial hash value from an actual hashing of the string. The function seedRNG() seeds the RNG. The function randInt() returns a random integer between 1 and *m* from the RNG. The returned value is the pseudo hash value. In the function testString(), testing of bits is explicitly halted at the detection of the first 0 bit. Compared to the method of Kirsch and Mitzenmacher, our RNG method requires only one actual hash and it can use a "good" LCG RNG algorithm (i.e., one with well-known properties) for generating the pseudo hash values. In our implementation of the RNG hash method we use the following LCG (from Jain [10]),

$$x_n = 7^5 x_{n-1} \bmod\left(2^{31}-1\right) \tag{2}$$

where x_n is the *n*th random integer value. We evaluate computation time and probability of false positive for both the Kirsch and Mitzenmacher, and RNG methods later in this paper.

5. Analysis of Best-of-N Method

In this section we derive an expression for probability of false positive for the *Best-of-N* method. Defining *S* as the random variable for the number of bits set in a Bloom filter; we derive expressions for mean and variance of *S*. Assuming a normal distribution of *S* and using order statistics, we get a computable expression for probability of false positive as a function of *N*.

The derivation of the probability of false positive of a regular Bloom filter (a *Best-of-N* Bloom Filter where $N=1$) is well described in the literature [1, 2]. We partially repeat this derivation below. We define a Bloom filter to have *m* bits and *k* hash functions per hash function group. The number of elements (file name strings in our case) represented in a Bloom filter is *n*. We assume that a hash function selects each array position with equal probability. Then, the probability, *p*, that a given bit is set in a Bloom filter is

$$p = \Pr[\text{a given bit is set}] = 1-\left(1-\frac{1}{m}\right)^{kn}. \tag{3}$$

A false positive occurs when a tested string that is not a member of the Bloom filter maps to *k* bit positions that are set (i.e., have been set by strings mapped into the Bloom filter). This event occurs as

$$\Pr[\text{false positive}] = p^k. \tag{4}$$

For a Bloom filter with *s* bits set, $p = s/m$ and thus

$$\Pr[\text{false positive}] = \left(\frac{s}{m}\right)^k. \tag{5}$$

The mean of *S* is the probability that a bit is 1 multiplied by the total number of bits,

$$E[S] = mp. \tag{6}$$

The variance of *S* requires the derivation of the second moment of *S*. Let U_i ($i = 1, 2, \ldots, m$) be the random variable that is set to 1 if bit *i* is set to 1 and 0 otherwise. Then $S = U_1 + U_2 + \cdots + U_m$ where

$$\Pr[\mathrm{U_i} = 1] = 1-\left(1-\frac{1}{m}\right)^{kn}. \tag{7}$$

Hence,

$$E[\mathrm{S}] = E[U_1] + E[U_2] + \cdots + E[U_m] = m\left(1-\left(1-\frac{1}{m}\right)^{kn}\right). \tag{8}$$

For the second moment,

$$E^2[S] = E[U_1 + \cdots + U_m]^2 = \sum_i E^2[U_i] + \sum_{i\neq j} E[U_i U_j] \tag{9}$$

Since $U_i^2 = U_i$, we already know that

$$E^2[U_1] + \cdots + E^2[U_m] = m\left(1-\left(1-\frac{1}{m}\right)^{kn}\right). \tag{10}$$

We notice that $E[U_i U_j]$ is equal to the probability that both bits *i* and *j* are set (this occurs when $U_i U_j = 1$). This is,

$$\begin{aligned}&\Pr[U_i U_j = 1] = \\ &1 - \Pr[U_i = 0] - \Pr[U_j = 0] + \Pr[U_i = 0 \text{ and } U_j = 0]\end{aligned} \tag{11}$$

which is

$$Pr[U_i U_j = 1] = 1 - 2\left(1-\frac{1}{m}\right)^{kn} + \left(1-\frac{2}{m}\right)^{kn}. \tag{12}$$

And thus we obtain $E^2[S]$ (13) directly. From (8) and (13) we can directly obtain the variance $\sigma^2[S]$ (14).

$$E^2[S] = m\left(1-\left(1-\frac{1}{m}\right)^{kn}\right) + m(m-1)\left(1-2\left(1-\frac{1}{m}\right)^{kn} + \left(1-\frac{2}{m}\right)^{kn}\right) \tag{13}$$

$$\sigma^2[S] = m\left(\frac{m-1}{m}\right)^{kn} + m^2\left(\frac{m-2}{m}\right)^{kn} - m\left(\frac{m-2}{m}\right)^{kn} - m^2\left(\frac{m-1}{m}\right)^{2kn} \tag{14}$$

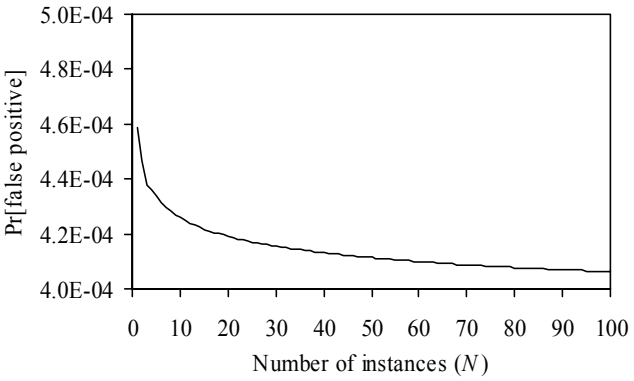

Figure 5. Probability of false positive for m/n = 16

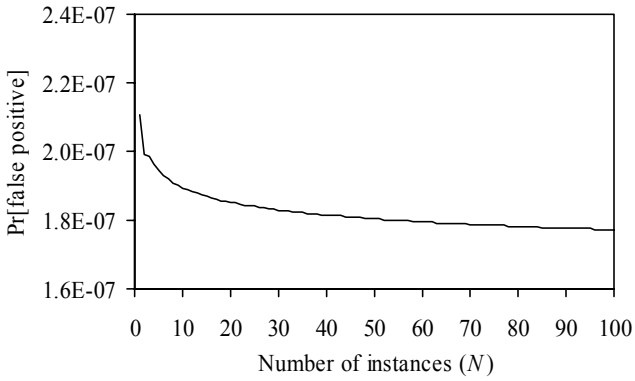

Figure 6. Probability of false positive for m/n = 32

We use order statistics [9] to determine the mean value of S given N samples (i.e., N instances of a Bloom filter) where the minimum value of samples $S_1, S_2, \ldots, S_N$ is selected as *Best-of-N*. For the random variable S with probability distribution function $f(s)$, cumulative distribution function $F(s)$, and N independent samples $S_1, S_2, \ldots, S_N$, the minimum value of the samples is the first order statistic,

$$S_{\min} = S_{(1)} = \min(S_1, S_2, \ldots, S_N). \quad (15)$$

For a continuous distribution,

$$f_{\min}(s) = N(1 - F(s))^{N-1} f(s). \quad (16)$$

The mean can be computed as

$$E[S_{\min}] = \int_{-\infty}^{\infty} s f_{\min}(s) ds. \quad (17)$$

Based on heuristic and empirical evidence, the distribution of S appears to be close to normal. Thus,

$$f(s) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(s-\mu)^2}{2\sigma^2}} \quad (18)$$

and

$$F(s) = \frac{1}{2}\left(1 + erf\left(\frac{x-\mu}{\sigma\sqrt{2}}\right)\right) \quad (19)$$

where $\mu = E[S]$ and $\sigma = \sigma[S]$. Substituting (18) and (19) into (16) we get,

$$f_{\min}(s) = \left(\frac{1}{2^{N-1}}\right) N\left(erfc\left(\frac{s-\mu}{\sigma\sqrt{2}}\right)\right)^{N-1}\left(\frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(s-\mu)^2}{2\sigma^2}}\right). \quad (20)$$

We can now give an expression for the probability of false positive of the *Best-of-N* method,

$$\Pr[\text{false positive}] = \left(\frac{E[S_{\min}]}{m}\right)^k \quad (21)$$

where $E[S_{\min}]$ is computed by substituting (20) into (17).

5.1 Numerical results

For a given m and n, the value of k to minimize the probability of false positive of a Bloom filter can be determined. The probability of false positive from (3) and (4) is

$$\Pr[\text{false positive}] = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{\frac{-kn}{m}}\right)^k. \quad (22)$$

The value of k that minimizes the exact expression in (22) can be solved for directly and is

$$k_{opt} = \frac{-\ln(2)}{\ln\left(\frac{m-1}{m}\right)n}. \quad (23)$$

The value of k that minimizes the approximate expression in (22) can be solved for directly and is

$$k_{opt} = \ln(2)\frac{m}{n}. \quad (24)$$

As m becomes large, the values of k_{opt} in (23) and (24) converge to the same. A table of false positive values for varying m, n, and k based on (24) is presented in [6].

For a given m and n where k is chosen optimally, we study the probability of false positive as a function of N. Figure 5 shows a plot of the probability of false positive (i.e., from (21)) as a function of N for $m/n = 16$ and $k_{opt} = 11$. Figure 6 shows the same plot for $m/n = 32$ and $k_{opt} = 22$. For Figure 5 $n = 1000$ and $m = 16{,}000$ corresponding to an almost 16 KByte Bloom filter representing 1000 strings (or 1000 files shared in our P2P application). For Figure 6 the same value of n is used and m is increased to 32,000. It can be seen that *Best-of-N* results in a reduction in probability of false positive. Figure 7 shows the improvement factor for $m/n = 8$, $m/n = 16$ (from Figure 5), $m/n = 32$ (from Figure 6), and $m/n = 64$. Table 1 summarizes the improvement for N = 2, 5, 10, 50, and 100 from Figure 7. It can be seen

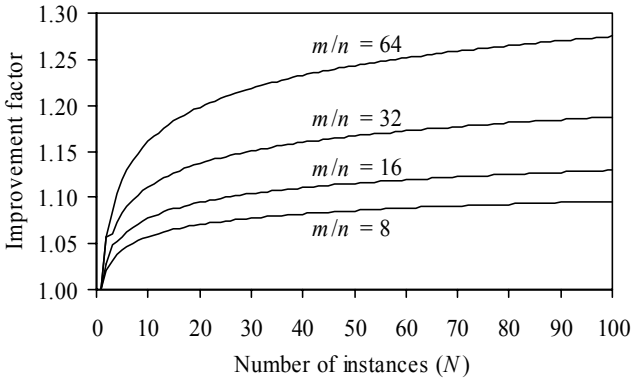

Figure 7. Improvement factor for various m/n

Table 1. Summary of Best-of-N improvement

N	$m/n = 8$	$m/n = 16$	$m/n = 32$	$m/n = 64$
2	1.021	1.028	1.058	1.057
5	1.043	1.058	1.083	1.119
10	1.058	1.078	1.111	1.161
50	1.086	1.115	1.167	1.243
100	1.096	1.129	1.188	1.275

that as m/n increases the improvement factor also increases. For $m/n = 32$ an almost 19% reduction in probability of false positive is achieved for $N = 100$.

6. Experimental Evaluation

In this section we compare the analytical model to a real implementation of the *Best-of-N* method for probability of false positive. We also evaluate the computation time when run on a typical desktop PC. An implementation of a Bloom filter with the *Best-of-N* method was written in C (the implementation is freely available from the authors by request). The following four hashing methods were implemented (input was a list of strings, described in Section 6.1):

- MD5 using the implementation from [4]. MD5 was chosen as a widely used and known hash function.
- CRC32 using an 8-bit table look-up implementation from [18]. CRC32 was chosen as a widely used and relatively efficient hash function to be implemented in software.
- Our new RNG hashing method from Section 4 using CRC32 for the seed hash.
- Kirsch and Mitzenmacher's method from [12] using CRC32 for the seed hashes.

In addition to the above four hashing methods used to hash and map real strings into a Bloom filter, a "perfect hashing" was implemented using an RNG to generate a random sequence of values. This perfect hashing served as a control to eliminate any effects from the real hashing methods that may be less than completely random.

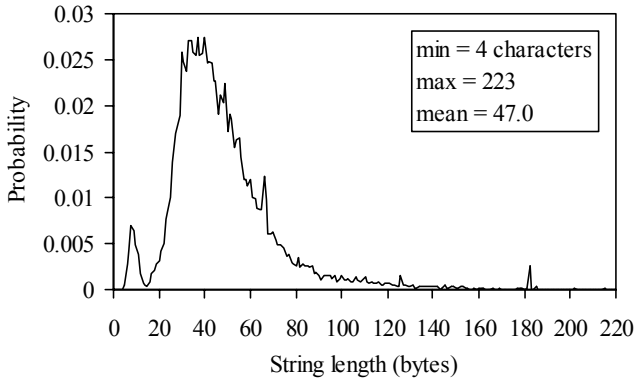

Figure 8. Histogram of string lengths for test set

One critical aspect of our experiments was to create N groups of hash functions based on one hash function implementation. For CRC32, we initialized the CRC accumulator with a different value each time a new hash function was needed. For the MD5 implementation, we added a different value at the end of the string to be hashed each time a new hash function was needed.

6.1 Description of the experiments

All experiments were executed on a Dell OptiPlex GX620 PC (Pentium4, 3.4 Ghz, 2 MBytes cache) with 1 GByte RAM with WindowsXP as the operating system. The gcc compiler (version 3.4.2 mingw-special from Dev C++ [15]) with no optimizations was used in all cases.

A list of 25,000 strings of unique music file names was obtained using Bearshare [14] (this list is freely available from the authors by request). The list of music file names was generated manually by searching names of artists and compiling a list of the songs retrieved by the queries. Each string consists of artist name and song title. This list of strings was used for the input to all experiments (except for the perfect hashing experiments where no strings were hashed). Figure 8 shows a histogram of string lengths from the test set. The mean string length was 47 bytes. Thus, with $m/n = 16$ there are 16 bits per string mapped into the Bloom filter resulting in a storage savings of over 20 times (i.e., $47/2 = 23.5$).

The two response variables of interest were:

- Probability of false positive for the Bloom filter.
- Execution time to generate a Bloom filter.

Probability of false positive was measured in two ways:

- Analytically by counting the number of bits set to 1 and using (5).
- Empirically by testing for membership with a list of test strings where none of the strings in the list were already represented in the Bloom filter.

Execution time was measured using C time functions with an accuracy of 10 ms on WindowsXP.

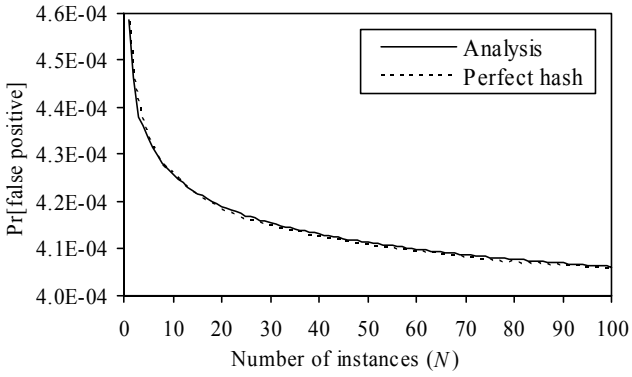

Figure 9. Results from false positive experiment #1

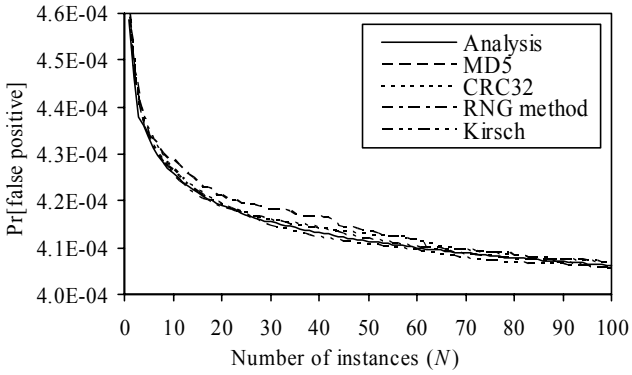

Figure 10. Results from false positive experiment #1

The control variables were:

- Hashing method used
- Bloom filter parameters m, n, and k
- *Best-of-N* parameter N
- Number of strings used in the string test set

Experiments were designed to evaluate the probability of false positive, (including comparison of the analytical model to actual implementation) and computation run time (CPU time). All experiments, unless otherwise stated, were executed using the four hashing methods and "perfect hashing" using an RNG. For all experiments n was set to 1000 (corresponding to a reasonable number of files shared by a P2P node) and m set to 16,000 corresponding to $m/n = 16$ (k was chosen optimally as $k_{opt} = 11$ from [6]). The experiments were:

False positive experiment #1: Vary N from 1 to 100. Measure probability of false positive using (5). Collect the mean from 10,000 iterations for each value of N.

False positive experiment #2: Repeat the previous experiment, except measure probability of false positive empirically. The test set of strings contained 20,000 strings.

Run-time experiment: Repeat the false positive experiment and collect the run time (CPU time) for each value of N.

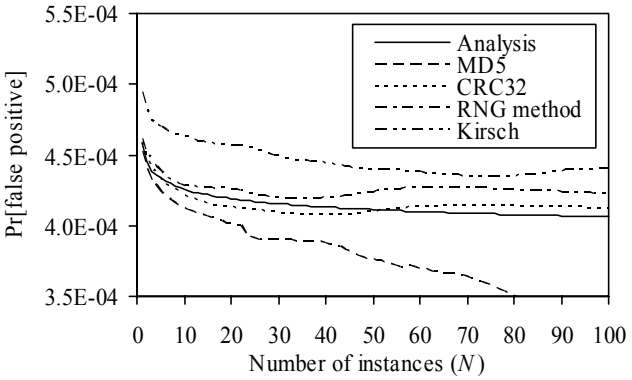

Figure 11. Results from false positive experiment #2

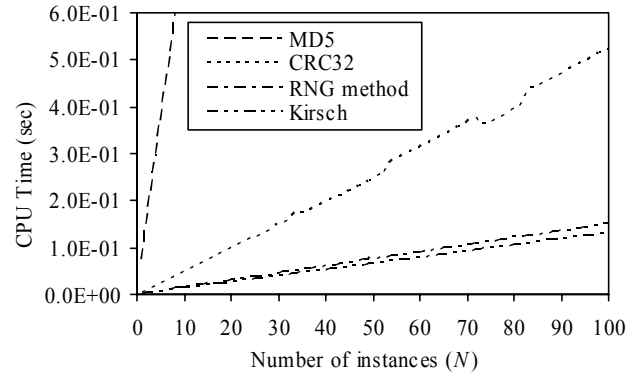

Figure 12. Results from run time experiment

6.2 Results from the experiments

The *false-positive experiment #1* results are shown in Figure 9 for perfect hashing and in Figure 10 using the four hashing methods studied. In Figure 9 it can be seen that the probability of false positive from the implementation and analysis agree perfectly (within 1% for all values of N). This validates our analytical model, including our assumption of normality for the number of bits set to 1 (S). Figure 10 shows that using a real, non-ideal hash function results in a probability of false positive with greater variability compared to using a "perfect" hashing function.

The *false-positive experiment #2* results are shown in Figure 11. This figure specifically shows a lower probability of false positive for our RNG method compared to the method of Kirsch and Mitzenmacher.

The *run-time experiment* results are show in Figure 12. The graph shows the CPU time to generate one Bloom filter using the *Best-of-N* method with the four different hashing methods. For our test PC we found that for $N = 1$:

- MD5 requires 74 milliseconds
- CRC32 requires 4.9 milliseconds
- RNG method requires 1.5 milliseconds

- Kirsch and Mitzenmacher method requires 1.3 milliseconds

These times increase linearly with *N*. Thus, the RNG and Kirsch and Mitzenmacher methods are about the same, CRC32 is 3 times greater in CPU time required than these two methods, and MD5 is 15 times greater.

7. Related Work

Bloom filters were first proposed by Bloom in 1970 as a space/time trade-off for hash tables [1]. Bloom filters reduce the space requirement for a hash table by allowing for a small probability of error – a false positive that a tested element is represented in the set. Bloom filters have found use in spell checkers, distributed databases, distributed caching, and in many other areas. A survey of network applications of Bloom filters is in [3].

Improvements to Bloom filters have been studied by many researchers. Counting Bloom filters were proposed by Fan et al. [7] (and improved by Bonomi et al. [2]) as a means of allowing insertion and deletion of elements (standard Bloom filters do not allow for element deletion). Lumetta and Mitzenmacher [13] applied the power of two choices to Bloom filters to reduce the probability of false positive. Lumetta and Mitzenmacher use two groups of hash functions for mapping elements and testing for element membership. This increase in processing results in a decrease in probability of false positive. The improvements reported are factor of 2 to 3 reduction in probability of false positive. However, the added expense in computation is not clear. We were unable to reproduce the Lumetta and Mitzenmacher results as we were unable to implement their method for testing element membership as described in [13]. Their work, however, was the primary motivation for our *Best-of-N* method and, as such, bears the closest resemblance to our work.

An optimal Bloom filter replacement was studied by Pagh et al. [17]. Their approach is to use dynamic multisets to reduce membership testing time and space usage (and thus also probability of false positive for a given space allocation). This work is theoretical with no reported experimental implementations or results.

For faster hashing for Bloom filters, Kirsch and Mitzenmacher [12] explored the use of pseudo hashing. Their work is described in Section 4 of this paper and is the motivation for our RNG hashing method.

8. Summary and Future Work

In summary, in this paper we have explored two improvements to Bloom filters:

- A new *Best-of-N* method that reduces the probability of false positive by generating *N* instances of a Bloom filter and selecting the best.
- A new RNG hashing method that generates pseudo hashes given a single seed hash.

The *Best-of-N* method was analyzed using a probabilistic analysis and order statistics, and evaluated as an actual implementation. It was shown that with very modest computation, the probability of false positive can be reduced by 10 to 20%. This is a new and useful contribution to the body of knowledge for Bloom filters.

Our application for the Bloom filter is a power management proxy for P2P file sharing applications. The Bloom filter is a space efficient mechanism for storing a list of files shared. We estimate that if only 25% of PCs running P2P applications contained SmartNICs with proxy capability, a savings of over $38 million per year in the US could be achieved (see appendix for calculations).

Future work includes fully implementing and evaluating the P2P proxy. We will also explore further improvements to Bloom filters including:

- Compare our *Best-of-N* method to Lumetta and Mitzenmacher's power of two method [13].
- Evaluate the normality assumption we made for the number of bits set in a Bloom filter (*S*).
- Explore if different values of *k* may improve the performance of the *Best-of-N* Bloom filter method.
- Analyze the probability of hashing collisions (and thus false positives) for our RNG hashing method.

Acknowledgment

The authors thank Dr. Kamran Sayrafian-Pour from the Advanced Network Technologies Division (NIST) for his multiple helpful comments that improved the quality of this paper.

Disclaimer

Certain commercial equipment, instruments, or materials are identified in this paper in order to specify the experimental procedure adequately. Such identification is not intended to imply recommendation or endorsement by the National Institute of Standards and Technology, nor is it intended to imply that the materials or equipment identified are necessarily the best available for the purpose.

References

[1] B. Bloom, "Space/Time Tradeoffs in Hash Coding with Allowable Errors," *Communications of the ACM*, Vol. 13, No. 7, pp. 422-426, 1970.

[2] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An Improved Construction for Counting Bloom Filters," *14th Annual European Symposium on Algorithms*, LNCS 4168, pp. 684-695, 2006.
[3] A. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey," *Internet Mathematics*, Vol. 1, No. 4, pp. 485-509, 2005.
[4] L. Deutsch, "MD5 Homepage," 2006. Available: http://userpages.umbc.edu/~mabzug1/cs/md5/md5.html.
[5] Energy Information Administration, "U.S Household Electricity Report," July 2005. Available: http://www.eia.doe.gov/emeu/reps/enduse/er01_us.html.
[6] L. Fan, P. Cao, and J. Almeida, "Bloom Filters - The Math," 2000. Available: http://www.cs.wisc.edu/~cao/papers/summary-cache/node8.html.
[7] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol," *IEEE/ACM Transactions on Networking*, Vol. 8, No. 3, pp. 281-293, 2000.
[8] M. Fetscherin and S. Zaugg, "Music Piracy On Peer-to-Peer Networks," *Proceedings of the 2004 IEEE International Conference on e-Technology, e-Commerce and e-Service*, pp. 431-440, 2004.
[9] R. Hogg and A. Craig, *Introduction to Mathematical Statistics*, Fifth Edition, Prentice-Hall, 1995.
[10] R. Jain, *The Art of Computer Systems Performance Analysis*, John Wiley & Sons, 1991.
[11] K. Kawamoto, J. Koomey, B. Nordman, R. Brown, M. Piette, M. Ting, and A. Meier, "Electricity Used by Office Equipment and Network Equipment in the U.S.: Detailed Report and Appendices," *Technical Report LBNL-45917*, Energy Analysis Department, Lawrence Berkeley National Laboratory, 2001.
[12] A. Kirsch and M. Mitzenmacher, "Less Hashing, Same Performance: Building a Better Bloom Filter," *Technical Report TR-02-5,* Computer Science Group, Harvard University, 2005.
[13] S. Lumetta and M. Mitzenmacher, "Using the Power of Two Choices to Improve Bloom Filters," unpublished, 2006. Available: http://www.eecs.harvard.edu/~michaelm/postscripts/bftwo.ps.
[14] Musiclab, BearShare P2P Application, 2006. Available: http://www.bearshare.com/.
[15] MinGW - Minimalist GNU for Windows, 2006. Available: http://www.mingw.org/.
[16] F. Oberholzer and K. Strumpf, "The Effect of File Sharing on Record Sales an Empirical Analysis," School of Business, University of Kansas, June 2005. Available: http://www.unc.edu/~cigar/papers/FileSharing_June2005_final.pdf.
[17] A. Pagh, R. Pagh, and S. Rao, "An Optimal Bloom Filter Replacement," *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 823-829, 2005.
[18] A. Perez, "Byte-wise CRC Calculations," *IEEE Micro*, Vol. 3, No. 3, pp. 40-50, June 1983.
[19] S. Saroiu, K. Gummadi, and S. Gribble, "Measuring and Analyzing the Characteristics of Napster and Gnutella Hosts," *Multimedia System*, Vol. 9, No. 2, pp. 170-184, 2003.
[20] US Department of Energy, Energy Efficiency and Renewable Energy, "Estimating Appliance and Home Electronic Energy Use," 2005. Available: http://www.eere.energy.gov/consumer/your_home/appliances/index.cfm/mytopic=10040.

Appendix – Energy Savings Calculations

This appendix contains energy use and savings calculations as referenced in the body of the paper.

A.1 PC powered on 24/7 in typical household

A single PC powered-on 24 hours per day, 7 days per week (such as a PC running a P2P application) will add about 10% to the electricity usage of a typical US household. We calculate this as follows. A typical PC consumes 120 W [20]. There are 107 million households in the US with a total power consumption of 1,140 billion kWh/yr [5]. Thus, the average household electricity use is 10,654 kWh/yr. A 120 W PC powered-on 24/7 consumes 1051 kWh/yr. Thus, the addition of a single PC powered-on 24/7 would add 9.8%, or roughly 10%, additional electricity usage to the typical household.

A.2 Active time of a P2P node

A typical P2P node will be downloading or uploading files only 1% of the time (and is thus idle 99% of the time). We calculate this as follows. There are 1 billion downloads per week from 9 million users online at any time [16]. Thus, the average P2P node transfers about 16 files per day. The average size of a file shared in a P2P node is 8 Mbytes [8]. If we assume a 1 Mb/s download rate as typical [19], then about 17 minutes per day are spent transferring files, which is about 1% of 24 hours.

A.3 Estimated energy savings from P2P proxying

If only 25% of PCs running P2P applications contained SmartNICs with proxy capability, a savings of over $38 million per year in the US could be achieved. There are approximately 60 million PCs in households in the US [5]. We will assume that 10% of these PCs run a P2P file sharing application. We will further assume that 25% of these PCs running a P2P application will use a SmartNIC with proxying capability. If proxying can reduce the fully-on time by 8 hours per day (this assume full use by a user during 16 hours per day and otherwise idle during 8 hours per night) from 120W in fully on to 10W in sleep state, then the savings in electricity consumed is $38.5 million per year (using $0.08 per kWh).