

Using Bloom Filters to Speed-Up Name Lookup in Distributed Systems

M. C. Little, S. K. Shrivastava, and N. A. Speirs

*Department of Computing Science,
University of Newcastle,
Newcastle upon Tyne, NE1 7RU, UK.*

Abstract

Bloom filters make use of a “probabilistic” hash-coding method to reduce the amount of space required to store a hash set. A Bloom filter offers a trade-off between its size and the probability that the filter returns the wrong result. It does this without storing the entire set, at the cost of occasionally incorrectly answering yes to the question “is x a member of S ?”. The paper discusses how Bloom filters can be used to speed up name to location resolution process in large scale distributed systems. The approach presented offers trade-offs between performance (the time taken to resolve an object’s name to its location) and resource utilisation (the amount of physical memory to store location information and the number of messages exchanged to obtain the object’s address).

1. Introduction

The object location problem is that of how to find a named object in a computing system for the purposes of accessing it. To achieve location transparency, a distributed system must provide an automatic mechanism for locating objects. The mechanism takes the form of a *naming service* (also called a *binding service* or a *location service*, depending upon the context in which the term is being used) that maps an object's name to its location. The actual mapping process is frequently multi-stage. For example, initially the system might resolve a user supplied (context sensitive) name to a globally unique identifier (UID), which is then resolved to the host name which is then further resolved into an address to which invocation messages could be sent. Naming services play a central role in computing system, so their performance and reliability is a critical factor in determining the overall performance of systems. Achieving good performance and reliability of naming services is difficult in large scale distributed systems with billions of objects scattered over nodes separated by large physical distances. We present a speed-up technique based on hashing that is applicable in any stage of name to location resolution process.

We use a simple example to motivate the reader and illustrate our approach. Fig. 1 shows a simple one level mapping scheme: the name server maps a name to its location. The client on node A contacts the service, and gets the location information (messages 1 and 2); it then sends its invocation to the object (message 3). We assume that the system has expanded and it is required to partition the name service over several nodes (say a few hundred). There could be any number of reasons behind this requirement: (i) the number of entries in the name service has grown to be so large that the central database is becoming a performance bottleneck; (ii) the single server is a central point of failure; (iii) the system spans several countries and each country wants to maintain a name service for 'its' objects. The client now needs to know to which of the name servers the

query should be sent. For example, assume that client is somewhere in Australia, and the name servers are in Sydney, Tokyo, Mumbai, New York,....., Rio de Janeiro. It would be preferable if the client is not forced to search the name servers (sequentially or in parallel) for name resolution. The problem is difficult to solve in situations where no useful information can be derived from the structure of the name itself (for example, the name is a UID).

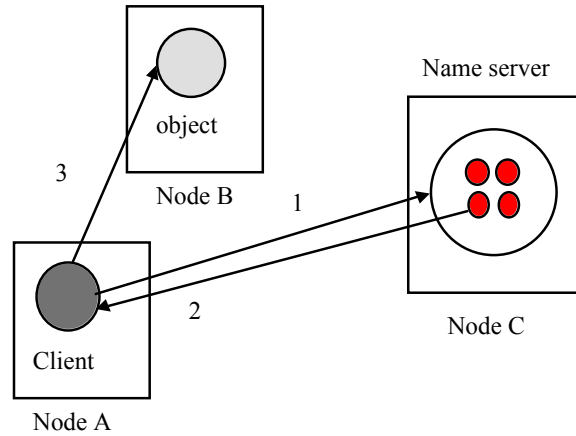


Fig. 1: One level naming service

Our scheme permits the client to maintain local *summary information* tables about each of the name servers, and to consult these tables to find the ‘right’ name server to contact (fig. 2, where h_1, \dots, h_n are the summary tables). Such summary tables can be used in a variety of ways. For example, the tables could be kept at each name server, rather than at a client; the client sends its query to any server (e.g., to the nearest server at Sydney in the example just considered) which then forwards the request if required. At an extreme, we could get rid of name server nodes all together, with each node maintaining name service for its local objects and a summary table for each of the other nodes in the system.

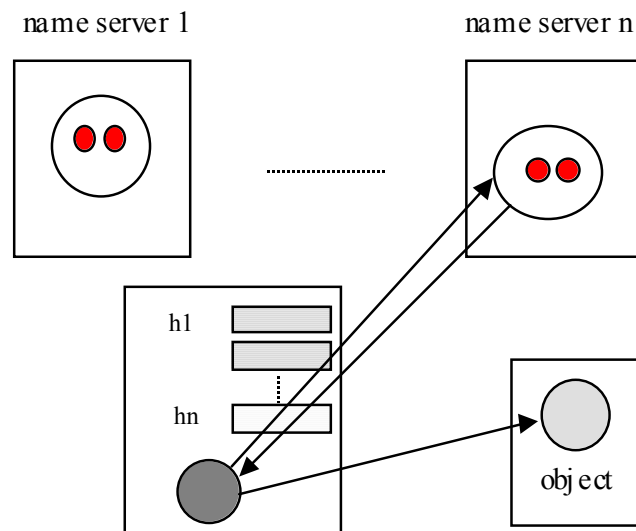


Fig. 2: Maintaining name server information

In the following sections we present a mechanism, Bloom filters, that offers trade-offs between performance (the time taken to resolve an object's name to its location) and resource utilisation (the amount of physical memory to store location information and the number of messages exchanged to obtain the object's address). This is possible because the algorithm the location mechanism uses is probabilistic, and may return an incorrect answer to the question "is X on machine Y?". However, as we shall show, given the right set of parameters, a probabilistic mechanism provides a good fit to the problems of managing a large name-space.

2. Bloom filters

In a traditional hash function, a hash area is organised into cells (the *hash table*), and an iterative process (the *hash function*) is used on input entities to generate hash addresses of the empty cells into which the entities can then be stored. Membership of a hash table is tested by a similar process of iteratively generating hash addresses of cells and comparing the contents with the test entities. Thus, a hash function requires space to store the keys to be hashed, and the entities to be stored.

Bloom filters were introduced in [1] as a "probabilistic" hash-coding method to reduce the amount of space required to store a hash set. They are primarily intended for applications where the majority of entities to be tested for membership will *not* belong to the given set. A Bloom filter offers a trade-off between its size and the probability that the filter returns the wrong result. It does this without storing the entire set, at the cost of occasionally incorrectly answering yes to the question "is x a member of s?" Such an occurrence is termed a *false hit*. If a false hit occurs, an alternate mechanism will have to be employed to determine whether the element is actually a member of the set. However, if the Bloom filter returns the right answer most of the time, then relying on a more expensive backup procedure in the (hopefully few) cases where the guess proves wrong can still yield satisfactory cost and performance.

2.1 Bloom filter Overview

A Bloom filter consists of a bit vector. To enter an item into a Bloom filter h hash functions are applied to the item and the corresponding bits in the vector are set to 1. The size of the filter is *not* proportional to the size of the entries to be "stored" within it, but on the number of bits to be set for each entry (this number is same for each entry). The number of bits to be set is used to determine the probability of a false hit. An item is a member of the filter *with a known probability* (determined by the filter's parameters and the total number of entries it is required to store information about) if all of the bits selected by the hash function have been set; if any of the selected bits is 0 then the item is definitely not a member of the set. To determine whether an item it truly a member of the set an alternate mechanism will be required.

The equations used to determine the filter size for a given false-hit probability are given in [1] and reproduced here:

$$N = a * (-\log_2 P) * (\log_2 e / (\log_2(1/T) * \log_2(1 - 1/T))) \quad \text{----- (1)}$$

where N is the size of the filter in bits, a is the number of entries to store into the filter, P is the desired false hit probability and T is the reciprocal of the proportion of bits still zero in the filter after entering a items; the optimal value of T is 2 as shown in [1]. The filter parameters can be used

to select a trade-off between error rate and filter size. For example, 2^{16} entries could be stored in a filter of approximately 300000 bits using 10 hash functions with an expected error rate of $1/2^{10}$.

To determine the number of hash functions to apply for each item (the number of bits to set in the filter):

$$h = \log_2 P / \log_2(1 - 1/T) \quad \text{----- (2)}$$

Bloom's equations assume that the hash function used to enter an item into the bit vector has equal probability of setting any bit. If this is not the case the actual false-hit probability may be higher than that predicted. See also [2] for additional discussion on Bloom filters.

2.2 Using Bloom filters in naming services

We describe a sample design for using such filters as part of a naming/binding service for distributed systems. We assume that the service has been distributed over a number of nodes, say, n ; every node has its own portion of the binding database, and in addition a separate Bloom filter for each of the n nodes. Every filter records the set of objects located on the machine it represents (see fig. 3). So for example, in a binding service for 100 million objects spread over 100 nodes, each node would maintain a set of 100 Bloom Filters each of which would maintain information for a million objects (assuming that the objects are uniformly distributed across all of the nodes).

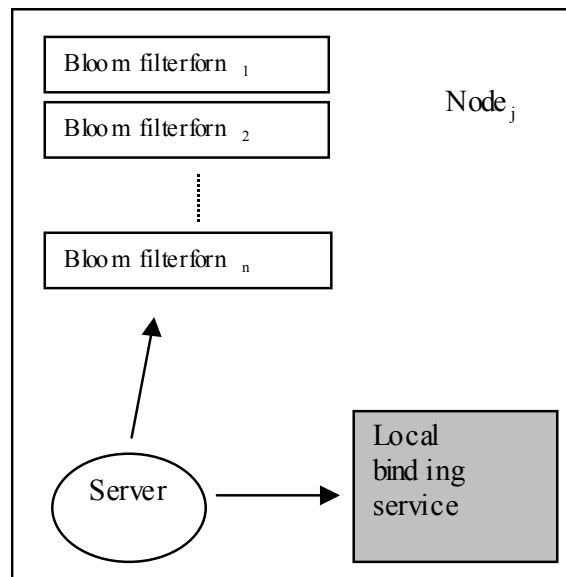


Fig. 3: local Bloom filters

The full course of events in resolving an object's name to its reference is fairly straightforward. A client directs its request to some preferred server. When presented with the object's name, the server sequentially searches its filters for a match. If a match is found, the server directs its binding request to that node (which could be itself). If this is a false hit, an error code will be returned and the server resumes its search amongst the remaining filters, and the procedure is repeated. If an entry has been made for that name then one of the hits is guaranteed to be correct. Alternatively, the client can search all the filters in a single go and simultaneously direct its request to the (small number of) hosts where the match is indicated. On the other hand, if no entry for that object has

been made then no match will be found. In this case an alternative location mechanism is required. This could for example be based on broadcast.

Obviously a Bloom filter must be populated and updated over time. This could either occur lazily, as servers discover the (new) locations of objects, or a background task could be responsible for propagating updated location information to each machine during periods of inactivity. Removing an item from a Bloom filter presents more of a problem: since a bit may be set in a filter as a result of an arbitrary number of inserted items, it is not possible to reset a bit to zero when an item is removed, as information about whether the item set each bit uniquely is not known. This means that over time a Bloom filter may accumulate garbage which cannot be reclaimed other than by generating a new filter. As with propagation of updates, this could occur as a background activity.

Table 1 below details the space/false hit trade-off for various distributed systems using optimal values for the number of hash functions. For example, from the first line of Table 1 we can see that 10^8 object entries could be stored in 100 filters (each representing a node in the system) of approximately 2.3Mbytes (i.e., a total of 230Mbytes for all filters) using 13 hash functions with an expected error rate of 0.009. In such a system, it would be practical to keep the entire set of filters in the main memory.

Objects	Nodes	Objects per Node	Hash fns	Filter Size (MB)	Bits in each	Pr(any 1 bit set)	Prob No false +ve in all machines
x	n	a = x/n	h	f	Filter b = $f \cdot 2^{13}$	$pb = 1 - (1 - 1/b)^{ah}$	$P = (1 - pb)^n$
1.0E+08	1.0E+02	1.0E+06	13	2.30	1.93E+07	0.49	0.991
1.0E+08	1.0E+03	1.0E+05	17	0.29	2.43E+06	0.50	0.992
1.0E+08	1.0E+04	1.0E+04	20	0.03	2.89E+05	0.50	0.991
1.0E+09	1.0E+02	1.0E+07	13	23.01	1.93E+08	0.49	0.991
1.0E+09	1.0E+03	1.0E+06	17	2.90	2.43E+07	0.50	0.992
1.0E+09	1.0E+04	1.0E+05	20	0.34	2.89E+06	0.50	0.991
1.0E+10	1.0E+02	1.0E+08	13	230.07	1.93E+09	0.49	0.991
1.0E+10	1.0E+03	1.0E+07	17	28.97	2.43E+08	0.50	0.992
1.0E+10	1.0E+04	1.0E+06	20	3.45	2.89E+07	0.50	0.991

Table 1: Bloom filter sizes for various system configurations.

In the table, pb corresponds to $1/T$ from the above equations, i.e., the optimal value of pb is $1 - (1 - 1/b)^{ah}$ and half the bits in the filter will be set to 1 after the filter has been populated. The values of f and h are tuned to produce the optimal value of pb. P in the final column represents the probability obtaining a single correct match, i.e., of looking at all filters and only obtaining a yes response from the filter for the node that is storing object location information. There is a

probability $(1 - P)$ of two or more filters claiming to host the object being sought. In this case all these nodes must be contacted. The figures in Table 1 show that this will happen less than 1% of the time for the given filter sizes.

To take a concrete example, imagine that we want to construct a naming service for a new generation of telephone system where users are given UIDs (personal identity numbers, PINs) in addition to telephone numbers, enabling callers to 'dial' UIDs. The mappings between UIDs to corresponding telephone numbers are maintained by this naming service. If a caller 'dials' the UID of the called party, the naming service is required to find the corresponding telephone number (possibly selecting one from a few, based on the preferences recorded by the called party) which the system then dials; therefore lookup times of less than a few hundred milliseconds on an average would be required. If we assume that 100 bytes of data are needed per user, then for a system containing 10 billion users, we need storage space of around 935 Giga Bytes. A globally accessible naming service will need to be designed and implemented. In its design, we need to minimise both client access time to the service itself and the actual name lookup time by the service. Hence, a single name server based solution is unlikely to meet the response time requirement; rather a physically distributed database will be required. Maintaining a fully replicated database over a few, say 100, geographically separated nodes (enabling a user to contact the 'nearest' replica) will reduce only client access times but not the lookup time; it also increases the total storage requirement by a factor of a hundred, thus does not appear to be an economical solution. Maintaining consistency between the replicas would also be a time consuming and expensive task. However, by partitioning the database over 100 nodes (each node configured as depicted in fig. 3), the size of each lookup database reduces to less than 10 Gigabytes. In addition, each node will have to maintain 100 Bloom filters taking a total of 23 Gigabytes of space. Such a scheme becomes practical in that it enables use of main memory databases, so name lookup times can be reduced drastically. Overall this Bloom filter based solution manages to reduce both client access times and name lookup times.

Note that a solution that does away with the need to contact the global binding service, and simply requires the client to broadcast the 'where' request to all the nodes in the system, relying on the 'correct' node to reply, is not appropriate here, as efficient network level broadcast facility scaling to billions of devices would be required. Such a solution on the other hand is frequently adopted in distributed operating systems (e.g., [3]), where nodes are connected by a LAN that supports broadcast. A broadcast based solution could be used amongst the small number of Bloom filter based name servers as the alternative lookup scheme to deal with the (hopefully rare) cases when the filters have not been populated.

2.3 Re-populating Filters and Garbage Collection

In many distributed systems objects are not static but migrate to different machines. In addition, objects must be able to leave the system when required and should be garbage collected. Garbage collection can be seen as a special case of object migration where the object migrates out of the system completely. Bloom filters can cope with object migration in the short term by updating the filter representing the new object location. However, as discussed in the previous section, it is not possible to delete the entry from the Bloom filter where the object originally resided. This means that whenever an object moves location a search of the filters will provide two object addresses and an overhead will be incurred in discovering the correct address. If object migration is infrequent,

hosts could maintain a list of object migrations to avoid this overhead. However, if object migration or deletion is common, the filters will need to be periodically repopulated from scratch to avoid such overheads. New filters can be transmitted between nodes and compression techniques are available to minimise the size of such messages [4].

A second reason for repopulating the filters is if the number of objects in the system increases substantially from the expected number. To minimise the number of false hits in the system, filters are sized so that half the bits in the filter will be set. If the number of objects in the system grows significantly after the filter size and number of hash functions has been determined, more bits in the filter will be set and the probability of a false hits will increase. The problem can be solved by re-computing the appropriate size of a new filter taking into account the additional number of entries.

It follows that repopulating filters is an important background task that has to be carried out. The frequency at which filters have to be repopulated depends upon the proportion of additional entries and the rate of object migration. In the following section we examine the overhead involved in repopulating filters.

2.4 Related Work

Application of Bloom filters has been investigated in text retrieval applications in the past [5, 6], but until recently, Bloom filters have not received attention within the context of distributed systems.

The use of Bloom filters has been proposed in a Web proxy cache sharing protocol [7, 8]. A collection of nodes (Web proxies) maintain Bloom filters recording information about each other; basically, each Web proxy maintains a Bloom filter for each proxy being shared, with each filter recording the list of URL's of the cached documents at that proxy. If a cache miss occurs at a proxy, that node searches its local filters to find if any other proxy has cached that document, so that it can fetch it from that proxy. In the absence of such filters, the node would be required to query the participating proxies for the document (possibly by a multicast), thus resulting in much increased network traffic. The number of Web proxies being shared in such applications is expected to be small (say between ten to hundred), and their work is focussed on limiting the number and size of inter-node messages. Although the Web proxy cache work is similar to ours, there are some significant differences. Our work is mainly motivated by the need to fragment a single name server database into smaller ones so as to optimise the lookup times. Furthermore we are aiming to keep the filters as small as possible whilst keeping the false hit probability small. In [4] it is shown that the size of inter-node messages may be reduced using compression techniques for Bloom filters and that this problem is orthogonal to that of minimising filter size.

The authors of [7, 8] also propose a probabilistic solution to the problem of re-populating filters. For each bit in the filter they store a count of the number of times each bit has been set. This allows them to delete items without having to re-populate the entire filter. They argue that a four bit counter for each bit in the Bloom filter will be extremely unlikely to overflow. However, even a four bit counter will immediately introduce an extra 400% overhead on storage. This is an excellent solution for systems with a relatively small number of objects and hence a modest memory requirement. However, it may become a very large overhead in the sort of systems that we are considering in this paper. As an extreme case consider the last entry in Table 1 where an extra

138Gb of memory would be required (4 bits for 10,000 filters of size 3.45 Mb). Hence in systems with a large number of objects where migration is infrequent we adopt the alternative solution of re-populating the filters as discussed in the previous sub-section. Our experimental work (section 3) concentrates on re-population times in addition to filter lookup times. Due to the nature of the application domain, neither of these issues are examined in [7, 8].

The OceanStore system [9] has developed what they term an attenuated Bloom filter that is used to provide a distributed routing service. An array of Bloom filters hold information and the i -th Bloom filter stores information for all nodes a distance i from the current node.

Design of scalable naming services has been researched extensively; the interested reader will find the review paper by Needham a good place to start [10]. The customary way of dealing with large name spaces has been to structure it into a hierarchy and use a multi-step resolution process as hinted earlier [11-14]. The approach presented here should be regarded as a technique available to designers for use in one or several stages of such a process, particularly when names are unstructured. For this reason we shall compare our solution against a traditional one-level naming service (like the one depicted in fig. 1).

3. Experimental validation

In order to determine the effectiveness of Bloom Filters in object location schemes, we have performed a few controlled experiments. We used a distributed environment based on the OMG's CORBA architecture [15], and chose to use the omniORB [16] implementation, which comes with a CORBA compliant implementation of the naming service (omniNames). We are able to show the benefits of being able to partition the naming service using Bloom filters.

3.1 Selecting a hash function

For a Bloom filter to work efficiently it is important that the probability of a false hit is low. As shown previously, the filter parameters can be used to select a trade-off between error rate and filter size. In addition, the accuracy of the Bloom filter relies critically upon how good the hash function is: the algorithms assume that each bit within a filter has equal probability of being set. The effects of a poor hash function may increase the number of false hits (and hence adversely affect the performance of the lookup mechanism) or may require the size of the filter to be increased in order to be able to reach the false hit probability.

A good hash function should provide a uniform distribution for entries, i.e., when the hash result is used to calculate hash bucket addresses, all buckets are equally likely to be picked. Similar hash keys should be hashed to different hash results: a single bit change in the key to be hashed should influence all bits of the hash result. In order to choose the best hashing algorithm for our requirements we examined the performance of a variety of those available [17-20], and from the results chose the PJW algorithm presented in [20].

3.2 Querying and populating the filter

In this section we shall describe how the hash function is used to enter an item into the Bloom filter, and can subsequently be checked for membership. The PJW hash function returns an integer value, which, combined with a deterministic random-number generator, is used to determine the bits to set within the vector for a given item.

The hash function we have chosen gives a good distribution of hash values for different keys. To verify how the algorithm performs within a Bloom filter, we conducted the following experiment: choosing the false hit probability of 0.01 we obtained the size of the Bloom filter from equation 1 for a varying number of entries (identified by a sequence of randomly generated numbers). From equation 2, 7 hash functions were required to be applied for each entry (i.e., 7 bits were to be set in the bit vector). New entries not already added to the filter were then created and tested for within it, and each false hit was recorded. From this, the false hit probability was calculated. This was repeated 100 times to obtain the average false hit probability and its standard deviation. The results are shown in figure 4.

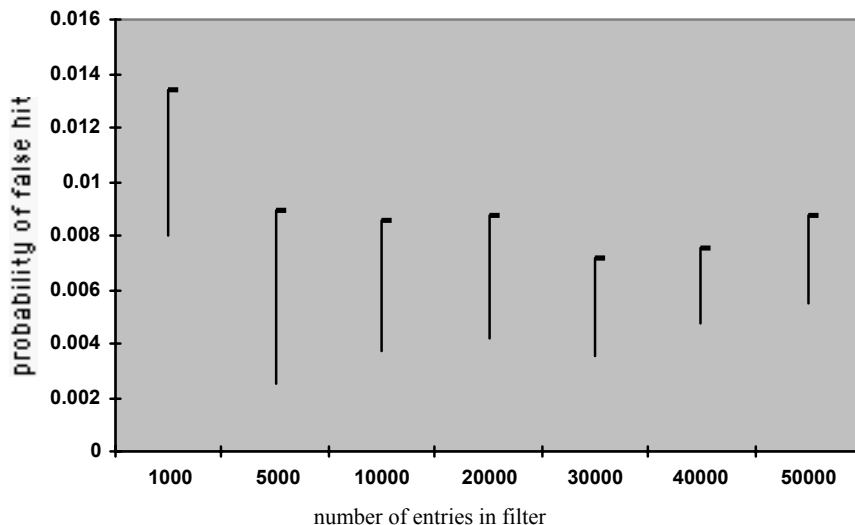


Figure 4: Bloom filter false hit probability (predicted 0.01).

As shown, the Bloom filter implementation performs well for the range of entries added. Although the false hit probability appears to be better than predicted by Bloom's equations we believe that this is the result of a relatively low number of executions.

Although a Bloom filter location service will typically only be populated during quiescent periods, the time taken to populate the filter is still important as shown in section 2.3. Therefore, we measured the average time taken to populate a Bloom filter, given an increasing number of entries. These experiments were performed using a lightly loaded Sun Ultra Enterprise 1/170 with 128M of main memory running Solaris 2.6. The results are shown in figure 5.

Figure 5 shows the expected result, that for a given false hit probability, the time taken to populate a Bloom filter is linear with respect to the number of entries, since it depends only upon the number of bits that must be set. Extrapolating from this graph suggests that a filter with 10^5 objects can be repopulated in approximately 1 minute. This is a relatively small system overhead for most systems except those where object migration is extremely frequent.

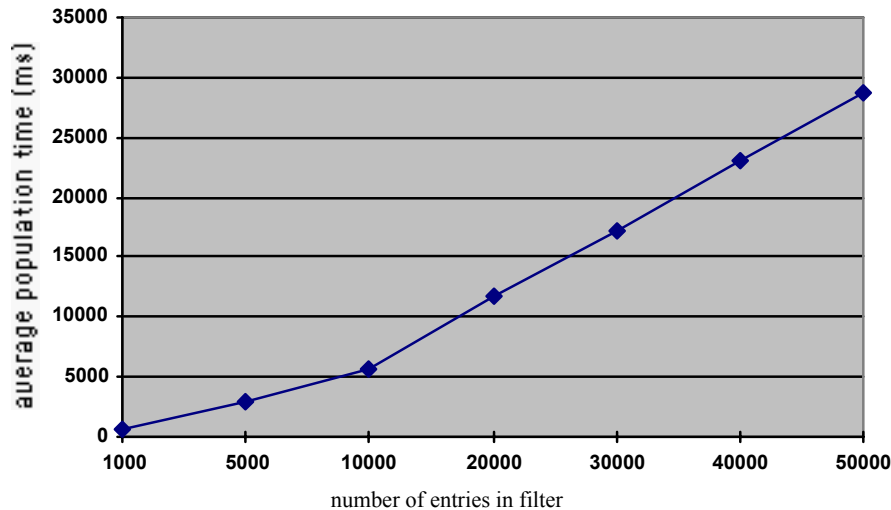


Figure 5: Bloom filter population time.

3.3 Performance results

We considered an object space consisting of 10^4 objects. We evaluated the performance of a traditional name service (like the one depicted in fig. 1) and its Bloom filter version (depicted in fig. 3), distributed over 8 machines. In this simple experiment, all the machines were over the same moderately loaded 10Mb/sec Ethernet LAN, where one-way process-to-process network communication latency, ∂ , was approximately 2ms. All the machines were lightly loaded Sun Ultra Enterprise 1/170 with 128M of main memory running Solaris 2.6.

3.3.1 Traditional name service

To determine the performance of the omniORB naming service (omniNames) we placed an increasing number of entries in it, and then measured the average time to locate a random entry. Figure 6 shows the performance measurements obtained for the name service lookup only, i.e., no remote invocation time is included. Note, we have no knowledge of the internal implementation of omniNames and hence cannot comment on why the naming service performance degrades with increasing population. However, within the population domain we shall examine for our Bloom Filter implementation (less than 1500) it is almost linear.

Looking at the message sequence of figure 1, we can see that the time to obtain an object's location information in a distributed environment will be $2\partial+t$, where t is the average name service lookup time. For example, taking the value of ∂ of 2ms, a name service populated by 10^4 entries retrieved data in 18ms (see figure 6) leading to a total time of approximately 22ms to provide an object's location.

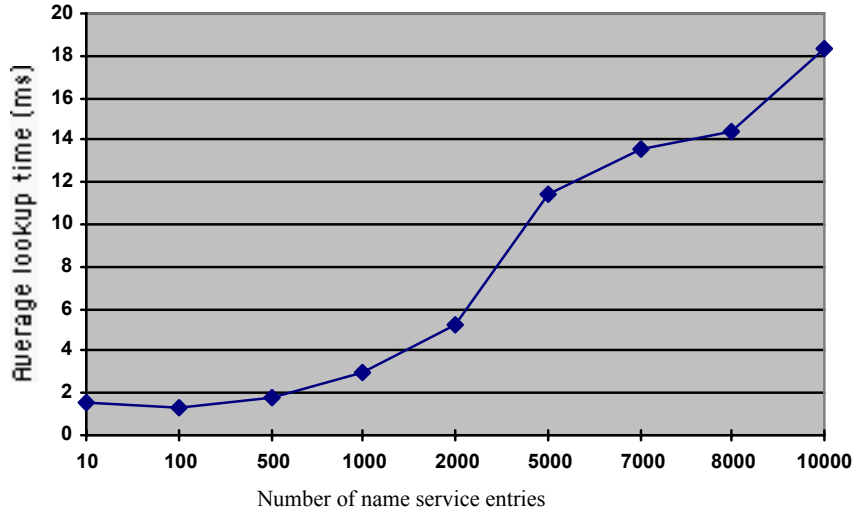


Figure 6: Average omniORB name service lookup time.

3.3.2 Bloom filter based servers

Assuming the arrangement depicted in figure 3, average lookup time for a Bloom filter location mechanism will be $2\partial+t'+\phi$, when no false hits are returned (this would be the case most of the time), t' is the name lookup time and ϕ is the time required to search the Bloom filter. Note that we expect t' to be much less than t , as the name service is managing a smaller number of objects ($1000/8 = 1,250$ entries in this particular case compared to 10,000 in the traditional name service). When false hits are returned, the time would increase to $(2\partial+t')*(1+\alpha)+\phi$, where α is the number of false hits. To arrive at the total lookup time for an object we must determine how long it takes to lookup an entry within a Bloom filter. Recall that there are 8 machines (and hence 8 filters) over which the objects have been evenly distributed.

Figure 7 shows the average lookup times for an increasing number of entries stored in the Bloom filter with a required false hit probability of 0.01. The higher plot is for the average lookup time for an entry that is present in at least one filter, and the lower plot is for the average lookup time for an entry that is not present in any of the filters.

On average the lookup times are constant, since the number of bits required to test for membership is also constant (7 in this experiment), and the average number of bits required to determine that an entry is not present is also constant (2 per filter). The two timings are similar because in both cases all 8 filters must be searched to obtain all “hits”, but most of the time a filter will only need to have 2 bits checked in order to determine that the entry is not present, i.e., having to test all 7 bits is an infrequent operation. From figure 6 the average name server lookup time t' for a server with 1250 entries will be 3ms. This gives the total lookup time in the normal case of no false hits to be about 8.3ms, an improvement by a factor of 2.65, and 15.3ms with one false hit.

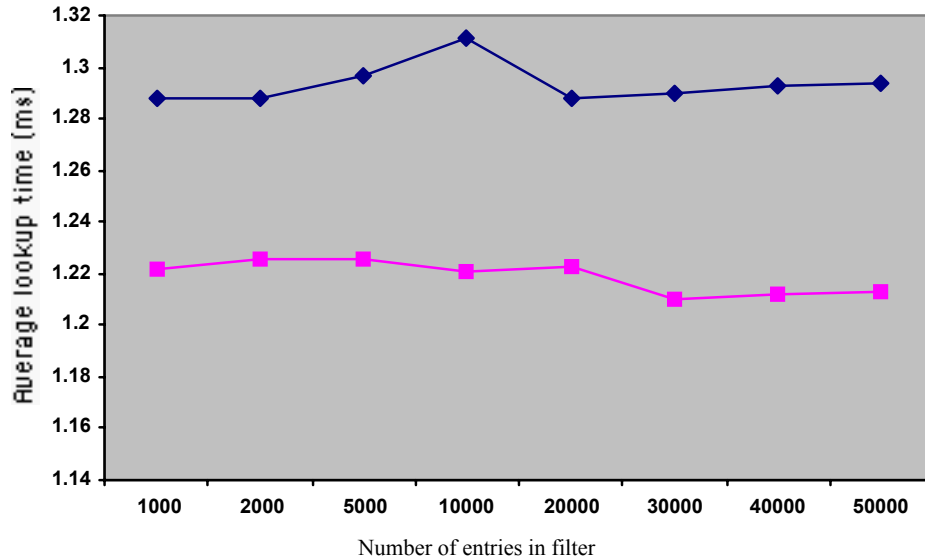


Figure 7: Average Bloom filter search time ϕ .

4. Concluding remarks

Achieving good performance and reliability of naming services is difficult in large scale distributed systems with billions of objects scattered over nodes separated by large physical distances. Within this context, we have described the use of Bloom filters in speeding up the name to location resolution process. We have described how to use such filters to obtain trade-offs between performance (the time taken to resolve an object's name to its location) and resource utilisation (the amount of physical memory to store location information and the number of messages exchanged to obtain the object's address).

Acknowledgements

This work has been supported in part by a research grant from BT, monitored by Paul Martin, BT, Adastral Park, Martlesham Heath. Irvin Shizgal drew our attention to the use of Bloom filters in naming systems.

References

- [1] Bloom, B.H. (1970) "Space/Time Trade-offs in Hash Coding with Allowable Errors", Communications of the ACM, 13(7) 422-426.
- [2] Mullin, J.K. (1983) "A second look at Bloom filters", Communications of the ACM, 26(8). 570-571.
- [3] Mullender, S.J. et al. (1990) "Amoeba: a distributed operating system for the 1990s", IEEE Computer, 23(5) 44-53.
- [4] Mitzenmacher, M. (2001) "Compressed Bloom Filters", 20th ACM Symposium on Principles of Distributed Computing (PODC-01), Newport Rhode Island, August 26-29, pp.144-150. ACM, New York.

- [5] Mullin, J.K. (1988) "Accessing textual documents using compressed indexes of arrays of small Bloom filters", *The Computer Journal*, 30(4), 343-348.
- [6] Shepherd, M.A., Phillips, W.J. and Chu, C.K. (1989) "A fixed-size Bloom filter for searching textual documents", *The Computer Journal*, 32(3) 212-219.
- [7] Fan, L., Cao, P., Almeida, J. and Broder, A.Z. (1998) "Summary cache: A scalable wide-area web cache sharing protocol", *Proceedings of ACM SIGCOMM'98*, Vancouver, British Columbia, August 31-September 4, pp.254-265. ACM, New York.
- [8] Fan, L., Cao, P., Almeida, J. and Broder, A.Z. (2000) "Summary cache: A scalable wide-area web cache sharing protocol", *IEEE/ACM Transactions on Networking*, 8(3) 281-293.
- [9] Kubiawicz, J. et. al., (2000) "OceanStore: An Architecture for Global-Scale Persistent Storage", 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-9), Cambridge Massachusetts, November 12-15, pp.190-201. ACM, New York.
- [10] Needham, R.M. (1993) "Names", In Mullender, S.J. (ed.), *Distributed Systems*. Addison-Wesley, Harlow, England.
- [11] Lampson, B.W. (1986) "Designing a Global Name Service", *Proceedings of 5th ACM Symposium on Principles of Distributed Computing*, ACM SIGACT-SIGOPS, Calgary, Alberta, Canada, August 11-13, pp.1-10. ACM, New York.
- [12] Sheltzer, A.B., Lindell, R., and Popek, G.J. (1986) "Name Server Locality and Cache Design in a Distributed Operating System", *Proceedings of 6th International Conference in Distributed Computing Systems*, Cambridge, Massachusetts, May 19-23, pp.515-522. IEEE Computer Society Press, Washington DC.
- [13] Welch, B., and Outerhout, J.K. (1986) "Prefix Tables: A Simple Mechanism for Locating Files in a Distributed System", *Proceedings of 6th International Conference in Distributed Computing Systems*, Cambridge, Massachusetts, May 19-23, pp.523-530. IEEE Computer Society Press, Washington DC.
- [14] Cheriton, D.R. and Mann, T.P. (1989) "Decentralizing a Global Naming Service", *ACM Transactions on Computer Systems*, 7 (2) 147-183.
- [15] Orfali, R., Harkey, D. and Edwards, J. (1996) "The essential distributed objects". John Wiley and Sons Ltd., Chichester, England.
- [16] omniORB2, Olivetti Research Laboratory. (<http://www.orl.com/>).
- [17] Knuth, D.E. (1973) "The Art of Computer Programming, III: Sorting and Searching", Addison Wesley, Harlow, England.
- [18] Wang, T. (1997) "Integer Hash Function", Concentric internal report, March 1997. (<http://www.concentric.net>).
- [19] Moreau, T. (1997) "The Frogbit Hash Function", CONNOTECH Experts-counsels Inc., May 1997. (<http://www.connotech.com>).
- [20] Aho, A., Sethi, R. and Ullman, J.D. (1986) "Compilers Principles, Techniques, and Tools". Addison Wesley, Harlow, England.