Scaling Filename Queries in a Large-Scale Distributed File System

Jonathan Ledlie, Laura Serban, Dafina Toncheva

January 14, 2001

Abstract

We have examined the tradeoffs in applying regular and Compressed Bloom filters to the name query problem in distributed file systems and developed and tested a novel mechanism for scaling queries as the network grows large. Filters greatly reduced query messages when using Fan's "Summary Cache" in web cache hierarchies[6], a similar albeit smaller, searching problem. We have implemented a testbed that models a distributed file system and run experiments that test various configurations of the system to see if Bloom filters could provide the same kind of improvements. In a realistic system, where the chance that a randomly queried node holds the file being searched for is low, we show that filters always provide lower bandwidth/search and faster time/search, as long as the rates of change of the files stored at the nodes is not extremely high relative to the number of searches. In other words, we confirm the intuition that keeping some state about the contents of the rest of the system will aid in searching as long as acquiring this state is not overly costly and it does not expire too quickly.

The grouping topology we have developed divides n nodes into $log(n)$ groups, each of which has a representative node that aggregates a composite filter for the group. All nodes not in that group use this low-precision filter to weed out whole collections of nodes by probing these filters, only sending a search to be proxied by a member of the group if the probe of the group filter returns positively. Proxied searches are then carried out within a group, where more precise (more bits per file) filters are kept and exchanged between the $\frac{n}{log(n)}$ nodes in a group. Experimental results show that both bandwidth/search and time/search are improved with this novel grouping topology.

1 Introduction

Centralized large-scale file systems like AFS, its successor CODA, and NFS have proliferated for two decades [8, 9, 19]. AFS systems at universities have scaled to 50,000 nodes or more. More recently, a drive to eliminate the bottlenecks imposed by centralized bookkeeping, lookup, and computation, has led to the development of decentralized systems that aim to scale to millions of nodes. The staple examples of this decentralization are Gnutella and Freenet, but more recently two other systems, CFS and PAST, have directly addressed scaling file distribution based on replication and intelligent hashing schemes [4, 5]. The prime drawback in these decentralized systems is that they lack a central, reliable source of information, either for access control, versioning, or name lookup. In this paper, we address several approaches to the name lookup problem in large-scale decentralized file systems.

Several studies motivate reducing bandwidth usage due to file name lookups in large-scale distributed systems. Two early studies, one aptly titled "Why Gnutella Can't Scale" [18, 20], underscore the difficulty and inherent infeasibility in searches in a network where a node has essentially no residual knowledge about its nearby nodes or the rest of the network. These papers and Ripeanu's empirical measurement of Gnutella [17] portray systems where nodes with low-bandwidth access not only are themselves swamped with queries, but act as anchors on the rest of the network as well. Gnutella, the most widely used distributed network currently in use, blindly floods the network with search queries, which go five or six hops away from the originator and then follow the same path back. Ritter and Sripanidkulchai show that networks on the order of thousands of nodes are enough to swamp a 56k modem. Tens or hundreds of thousands would exceed the capacity of much wider connections.

The newer distributed file systems, CFS and PAST, focus on data access, not on data lookup. Both view their systems as collections of multiple read-only file systems, where few authors are the only modifiers of data. The search problem existent in Gnutella, and other highly disparate distributed file systems of the same generation, maps entirely onto these new systems: there are still large numbers of nodes and no good mechanism for them to locate information based on file names without a centralized repository of information. In both SOSP 2001 papers on CFS and PAST, the authors leave searching as an open problem for future work.

We propose applying Bloom filters[2] to the problem of searching in a large-scale distributed file system. One mechanism would distribute a filter from every node to every node, and would lead to reduced bandwidth consumption compared to querying everyone and to more pinpointed (and therefore faster) searches; this would be at the cost of storage of the filters at each node that would scale linearly in the number of nodes. At the price of more storage used at each node, even fewer query messages would be necessary if this mechanism employed Compressed Bloom Filters instead of regular ones [12]. A second mechanism would develop a hierarchy of filters, where each node would only store a summary filter from each subgroup of the system, and then only direct queries to nodes in this subgroup if its filter provided a match. One could imagine this hierarchy extending several levels as the number of nodes increased. Other filtering mechanisms and topologies clearly exist.

We have developed a testbed that emulates a distributed file system in both fully-connected and grouped topologies. We have experimented on this system with a variety of filter sizes and with different flavors of compression. We have found that almost any filtering mechanism is superior in terms of bandwidth per search and speed of results to naively querying all nodes when the contents of the nodes are not unrealistically dynamic. Further, we show that grouping, even when the subgroups are randomly constructed, beats fully-connected filtering. We also demonstrate that the time for compression can make using Compressed Bloom Filters significantly slower than non-compressed filters and is highly dependent on the underlying compressor.

The remainder of this paper proceeds as follows: in section 2, we look at distributed file systems and at other recent uses of Bloom filters; in section 3, we go into more detail into the theory behind the tradeoffs in using different types of Bloom Filters and Compressed Bloom Filters and we discuss how group filters work; in section 4, we describe the implementation of our testbed system and the protocol used for grouping; section 5 examines our experimental results; section 6 discusses future directions for the project and concludes.

2 Background

2.1 Distributed File Systems

Gnutella and Freenet are successful, working distributed file systems that do not suffer from the constraints of centralization, like Napster, NFS, and AFS [7, 3]. Both use a "hop-based" approach to handle queries, where a node directly queries its neighbors which then forward the request to their neighbors, and so on. If one of these neighbors is slow or congested, then the search is slow. If no node on a particular path away from the originator has a match for the query, then all of the nodes on this path have been unnecessarily interrupted from handling other queries or performing other activities.

Freenet has two main enhancements beyond Gnutella:

1. It gives each object a unique identifier.
2. It caches search results on their way back to the search's originator.

What it does not provide – and is exactly the same problem that maps on to the newer distributed files systems of CFS and PAST – is a good mechanism to locate unique identifiers in the network. All three systems essentially hash names to unique identifiers but none provides a rapid, low-bandwidth search.

PAST and CFS differ primarily in their replication scheme both to allow quick access to data through locality and to grant reliability as nodes enter and leave the network [5, 4]. Analogous to the primary difference between AFS and NFS, PAST copies whole files and CFS distributes block-by-block. CFS has the advantage of parallel download of different parts of the same file from different nodes. As noted above, both leave name searching as an open problem.

2.2 Related Work

Databases have used Bloom filters to make searches faster since the early 1980s; R*'s distributed join algorithm uses them, for example [11]. More recently, they have been used in two research systems projects, one of which is currently in real-world use.

Fan's "Summary Cache" is a method for reducing the number of search queries in hierarchies of HTTP web cache storage servers [6]. Without summary caches, servers would query all nodes in their hierarchy and wait for a response from each. Since any number or even all of these responses could be negative (the file in question had not previously been stored in the hierarchy), hierarchies of caches did not scale well. Summary caches are Bloom filter summaries of the contents of each member of the cache hierarchy. Because Bloom filters never generate false negatives, there is no need to query caches whose filters show that they do not contain the file. With summary caches, Fan was able to reduce the number of intercache protocol messages by an order of magnitude and reduces the bandwidth consumption by over 50%. Summary Caches have become a part of the Squid Web Proxy cache that is used at many university and corporate gateways.

Like CFS and PAST, OceanStore is another widely distributed file system [10]. It focusses on data protection and availability through redundancy and cryptographic techniques, aiming to provide these through pro-active movement and caching of data before network problems occur. OceanStore uses "attenuated Bloom Filters" to perform local (searches of nearby nodes) quickly and then

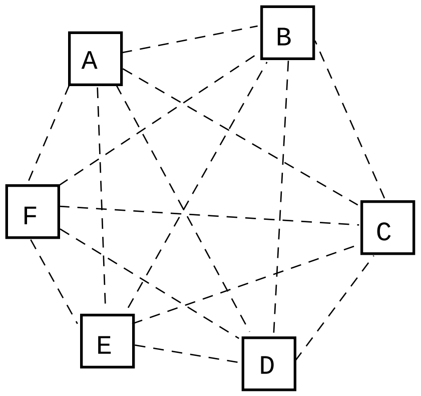


Figure 1: Fully connected search network Squares represent independent nodes. Dashed lines represent node-sized filters (filters based on a function of the number of files stored at a node), which are transmitted in both directions along each line (e.g. B $\rightarrow$ D and D $\rightarrow$ B).

falls back to a slow but reliable hierarchical method if this fails to produce results. Similar to our grouping method which performs a logical *OR* on a distinct subgroup of its neighboring nodes, the attenuated Bloom filter describes each directed edge in the network. OceanStore does not have the concept of hierarchies of filters with increasing reliabilities or of representative nodes that contain their subgroups filters; all nodes are representatives in OceanStore.

Other schemes being developed seek to optimize searches for peers that exhibit locality of interests [21]. Using their idea, a query would be categorized and then sent to a part of the network that had a high concentration of nodes that were also interested in this category. Sripanidkulchai's proposed solution works on top of existing protocols like Gnutella, Chord, and Pastry, and associates lists of peers who share the same interests. Like us, they argue that for scalability, it is impossible to maintain up-to-date state for all peers. Their project is still in development and it did not have published results at the current date. Our filters and particularly our grouping and subgrouping would interface well with their locality in interests idea.

3 Query Mechanisms

Our model of a distributed file system allows every node direct communication with every other node. This models the environment supplied by both CFS and PAST, where every node can "mount" the file system of every publisher and then query it directly. Of course, the underlying network, usually IP, does not supply direct connections among all nodes.

Because the nodes are fully connected, they can directly probe each other with queries and each node can send every other a summary of its contents. Figure 3 shows the paths for queries and filters in a complete graph with six nodes. Because the number of edges at any given time is $\frac{n(n-1)}{2}$, which is quadratic in the number of nodes, trying to propagate messages to all nodes (or even a small fraction) as the number of nodes gets large leads to difficulties like router buffer overflows and low response times.

In our analysis of using Bloom filters to improve queries in distributed systems, we have first compared the benefits of when filters are distributed to all nodes and when queries can pass among all nodes. After comparing the naive approach of sending a query to all nodes we are connected to (which in our implementation, was all of the nodes in the system) with using two filtered approaches, normal and Compressed Bloom Filter, we examine an application of composite filters that describes roughly the contents of a group of nodes.

3.1 Naive Filtering

The simplest to implement and visualize, "naive" filtering means no filtering at all: when a node performs a search, it contacts every node it can. While this approach may work for extremely dynamic systems where any summary information that would assist in improving search accuracy would expire immediately, it has the drawback of being both high in bandwidth consumption (many messages are sent per search) and slow. If files are evenly distributed and if the file a node is searching for actually exists in the system, a search will yield a positive result only after contacting half of the nodes, on average.

3.2 Bloom Filters

A Bloom filter is a quick and space-efficient data structure for representing a set of n elements to support membership queries. To represent a set $S = \{s_1, s_2, \ldots, s_n\}$ of n elements a Bloom filter uses an array X of m bits and k independent hash functions, $h_1, h_2, \cdots, h_k$ with range $\{0, 1, ..., m-1\}$. Initially, all the bits of the array are set to 0. An element s of S is included in the Bloom filter by setting each of the bits $h_i(s)$ to 1 for $1 \leq i \leq k$. To verify if an item x is in S the bits with indices $h_i(x)$ for $1 \leq i \leq k$ in the array X are checked. Clearly, if at least one of them is 0, x cannot be a member of S. If all of them are set to 1, x is assumed to belong to S. However, this assumption is incorrect with a certain probability since the same bit could be set to 1 for multiple items. That is, a Bloom filter may generate false positives, where it indicates that an element is in the set even though it is not.

Since in our design each node maintains a local Bloom filter to represent its own file system, changes of the set S must be supported [6]. This is achieved by maintaining for each location bit in the Bloom filter a count of how many times that bit was set to 1 (i.e., the number of elements of S that hashed to that bit position under the collection of hash functions used). This array is conventionally said to contain the bit's *phase*. All the counts are initialized to zero. Whenever a file x is added or removed from the file system of a node, k counts corresponding to the bits with indices $h_1(x), h_2(x), \cdots, h_k(x)$are incremented or decremented, respectively. When a count changes from 1 to 0, its corresponding bit is turned off since all the files that hashed to that bit had been removed. In addition, we maintain a saturation variable to keep track of the total number of bits that are set to 1 in the Bloom filter. If the number of files at a node increases, we expect the saturation count to approach the length of the filter in bits. When the filter becomes saturated, i.e., a majority of the bits are 1, the false positive rate increases, and the filter must be re-created to accommodate the increase in the number of files at the node. In our simulation, Bloom filters are regenerated when the saturation variable exceeds a given percentage of the filter's capacity.

It is useful to notice that in the Bloom filter data structure there is a clear tradeoff between m, the amount of memory used to represent the set S, and the probability of a false positive, f. Assuming that the hash functions used are random, after inserting n keys into a table of size m, the probability that a particular bit is still 0 is exactly:

$$\left(1-\frac{1}{m}\right)^{kn} \approx e^{\frac{-kn}{m}}$$

Letting $p = e^{\frac{-kn}{m}}$, the probability of a false positive in this situation is

$$\left(1-\left(1-\frac{1}{m}\right)^{kn}\right)^{k} \approx \left(1-e^{\frac{-kn}{m}}\right)^{k} = (1-p)^{k}$$

According to the analysis in [14] and [16], the optimal number of hash functions that minimizes the false positive rate above for a given size m of the Bloom filter is given by

$$k = (ln2)\left(\frac{m}{n}\right)$$

In this case the resulting minimum false positive rate f equals

$$f = \left(\frac{1}{2}\right)^{k} = (0.6185)^{\frac{m}{n}}$$

Pictorially, the false positive rate follows an exponential curve of the form in Figure 2.

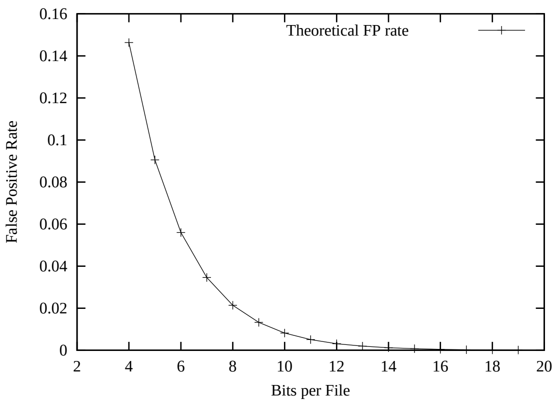


Figure 2: Theoretical False Positive rates for increasing bits per file, $\frac{m}{n}$

Clearly, if Bloom filters can be efficiently distributed and stored, their use will preempt many unnecessary query messages from ever leaving the origin of the search.

3.3 Compressed Bloom Filters

As Mitzenmacher suggests in [12], Compressed Bloom filters may be more appropriate in situations when the filter is not only a data structure used to summarize information at the nodes, but also a message that is passed between the nodes in order to support updates in a dynamic system. By using Compressed Bloom filters, nodes can reduce the number of bits broadcast, the false positive rate, and/or the amount of computation per look-up. The main cost of filter compression is the increased memory requirements at the end nodes that must process the larger uncompressed version of the filter. Additionally, the end points must compress and decompress the transmitted filter, thus ensuring additional processsing requirements.

The optimization problem for Compressed Bloom fiters can be cast in two ways. First, in parallel to the regular filters optimization problem, in the case of Compressed Bloom filters, one can also optimize for the false positive rate given a constraint on size, i.e., the number of transmission bits. That is, m and k can be chosen to minimize the false positive rate subject to a constraint on the size of the compressed/transmitted filter, z. If p denotes as before the probability that after n insertions a particular bit is still 0, the expected size of the Compressed Bloom filter is $mH(p)$, where $H(p) = -p\log_2(p) - (1-p)\log_2(p)$ is the entropy function.

According to the analysis in [12] the number of hash functions that minimizes the false positive rate for an uncompressed Bloom filter maximizes the false positive rate when the filter is compressed. More technically, subject to the constraint $m \times H(p) \leq z$ the expression defining the false positive rate $f = \left(1 - \exp^{\frac{-kn}{m}}\right)^{-k} = (1-p)^{(-\ln p)\left(\frac{m}{n}\right)}$, where $p = e^{\frac{-km}{n}}$, achieves a global maximum for $p = \frac{1}{2}$, or equivalently for $k = (\ln 2)\left(\frac{m}{n}\right)$. It can be shown that given a number of transmitted bits per entry $\frac{z}{n}$ and the contraint $m = \frac{z}{H(p)}$, minimizing f is equivalent to minimizing the expression $\alpha(p) = \frac{p}{\ln(1-p)} + \frac{1-p}{\ln(p)}$. The derivative of $\alpha(p)$ becomes 0 when $p = \frac{1}{2}$, is negative for $p \leq \frac{1}{2}$ and positive otherwise. That $p \leq \frac{1}{2}$ implies $k \leq \ln 2(\frac{m}{n})$ indicating that Compressed Bloom filters achieve a smaller probability of false positives by employing a smaller number of hash functions than the optimal number of hash functions for regular Bloom filters that use same number of transmitted bits per entry.

Alternatively, for a given false positive rate one can optimize for the compressed size z. Asymptotic analysis shows that the theoretical size of a compressed filter achieving the same false positive rate as a regular Bloom filter approaches $z = m \ln 2$, where m is the of the size of the standard filter.

In brief, theoretical results suggest that compression can be used to improve performance in a distributed system by reducing the false positive rate for a given compressed size and by reducing the transmission size for a given false positive rate. In addition, Compressed Bloom filters use smaller number of hash functions, which could potentially decrease the amount of processing per lookup.

As suggested in [12], arithmetic coding has been used to compress filters. The choice of arithmetic coding is natural since this scheme achieves “near-optimal compresssion with low variability” in fitting with the theoretical analysis which assumes that optimal compression is feasible. We used a publicly available adaptive arithmetic compressor implemened by Carpinelli, Moffat, Neal, and Witten [13]. The compressor was run with default parameters and the bits option on.

Similar highly compressed filtering mechanisms exist and would be interesting to try on the same problem. Lossy Dictionaries, for example, weigh each member of the set S, and uses a greedy algorithm to build a dictionary of maximum weight given constraints on space[15]. The dictionary consists of two tables of equal length. The keys in the set S are hashed to a cell value in one of the tables, and a union-find data structure is used to solve collions in an optimal manner. To verify if an element belongs to the set S, at most two cells of the dictionary must be checked. Since the data strcture requires at most two memory accesses per query, Lossy Dictionaries may be more time efficient than Bloom filters. However, the construction and updating of this data structure are not trivial, and may impose additional time requirements. In addition, Lossy Dictionaries imply a small rate of false negatives (i.e., negative response for an entry in the set), which may not be a desirable feature for some systems.

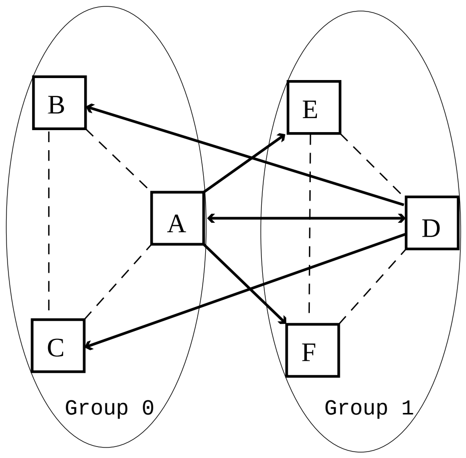


Figure 3: Group-based search network Squares represent independent nodes. Dashed lines represent node-sized filters. Thick lines are group filters and travel in the direction of the arrow. Ovals represent groups. A is the representative of group 0; D of group 1.

3.4 Aggregate Filters

When we began looking at having every node have a filter from every other node, we immediately recognized the problem that in order for filtering to truly scale it had to require less than quadratic communication among the nodes. By grouping nodes and then sending less precise filters which describe all of the files in these groups, we believed that less overall bandwidth and per-node storage would be used at the expense of some complexity. What we describe and what we have tested are two-tiered aggregate filters, but the reader can extrapolate that a similar grouping system would work recursively.

Aggregate filters are the “logical or” of all of the filters in a group. A probe against an aggregrate filter shows a match in the group with high probability, but, obviously, it cannot also tell which member of the group contains the real match. The group size we use in our experiments is $log(n)$ of the nodes.

If group filters are less precise or if a content-based grouping scheme is used as described later in this section, each node will use less storage with group filters than if it stored a filter for everyone, even if it stores precise filters of its immediate neighbors. Figure 3 portrays the topology of aggregated filters. In it, A and D are *representative* nodes that receive inter-group sized filters from the nodes in their group. The size of intergroup filters depend on the total number of files *in the group* $\times$ the group bits per file rate, which can be less that the intragroup size, in order to generate smaller, less precise intergroup filters. For example, A, B, and C all contain some number of files and communicate among themselves about how large to make their intergroup filter. B and C send A a Bloom filter (possibly compressed, as it will be mostly empty), A then *OR*s these filters with its own of the same size and send it to any requestors that would like a summary of the groups contents. A, B, and C all exchange filters like in a microcosm of Figure 3. Because the cardinality of the subgroups are substantially less than the total number of nodes, far fewer filters need to be exchanged. Where we had $\frac{n(n-1)}{2}$ filter messages before, with log(n) groups there are:

$$\frac{n}{log(n)}\left(\frac{log(n)(log(n)-1)}{2}\right)+$$
$$log(n)(n-log(n))+(n-log(n))(log(n))$$

where the first term is the number of aggregate intergroup messages, the second is the number of intragroup messages, and the third is the cost of group members sending the representatives their to-be-aggregated filters. Looking at how this grows with the number of nodes, we see:

Nodes	Fully Connected Edges	Grouping Edges
1,000	499500	19742
10,000	4999950000	3321392
1,000,000	5×10^{11}	39862362

Further subgrouping through recursion would reduce the number of messages even more.

4 System Design and Implementation

Each node in the system is an independent Java process consisting of four threads. Because they are separate processes, they can run on separate machines, ideally letting the tests scale to many (i.e. 1000) nodes. The component threads of the system are:

Query This thread waits for a random number of milliseconds based on an entry in the configuration and then chooses a random file from the domain of all possible files (also part of the configuration) to search for. Described in more detail in Section 4.1, the thread probes the local cache of neighbor's nodes for matches. If any are found, it creates a Search object and associates with it any neighbors (or groups) whose filters said they matched. For "naive" filtering, all filters match. The Query thread then initiates the search by sending out a *Verify* message to the first neighbor or group that matched. If none matched, it chooses another file (which is does not already have within its local file system), and begins probing again. The Search object is then added to the list of ongoing searches, and the thread goes back to sleep. Note that because the Query thread only initiates searches and these are then completed by the Protocol thread, there can be multiple searches and proxied searches occurring concurrently at the same node. In the experiments, we saw many searches taking over one second to complete, although they were beiing generated at a constant rate of approximately one per second.

File system changer This thread waits for a (different) random number of milliseconds specified in the configuration and then updates (adds or removes) a file from the node's "shared" files. It rebuilds the copy of the node's filter (based on the bits changed and phase) and adds a new entry to the list of filter deltas, noting the (possibly zero) bits changed and a timestamp for the action. This timestamp is used in the filter deltas, described in Section 4.2.

Protocol Server This thread functions as a UDP server, listening for protocol messages, responding to them, and then resuming listening. The actions it takes are outlined in Section 4.3. It serves to send neighbors any of this node's filters, to ACK or NACK query verify requests, and to proxy intergroup searches to other nodes in the same group, using its more precise intragroup filters.

Cache Refresh This thread looks at the caches of neighbor, group, and possibly representative filters and, if any are significantly out of date (null in our case), sends the node a request for its filter. It is primarily used to bootstrap the system and sleeps when all the nodes are up and the filters have been distributed.

The system also consists of two extra processes: one bootstrapping *Configurator* and one *Logger*. The Configurator supplies a stable base from which any node can discover the parameters for a particular experiment (e.g. whether to use compressed deltas). It is identified by its IP address and port, as are all the nodes. The Logger sits waiting for Log messages about the events in the system to arrive from the nodes and aggregates them.

The code is approximately 5700 lines of Java split up into 35 classes.

4.1 Filter Implementation

The implementation of the Bloom filters is based on the analysis in [16]. Ramakrishna suggests using Universal hash functions of the form:

$$h_{c,d}(x) = ((cx + d) \bmod p) \bmod m, \; and$$

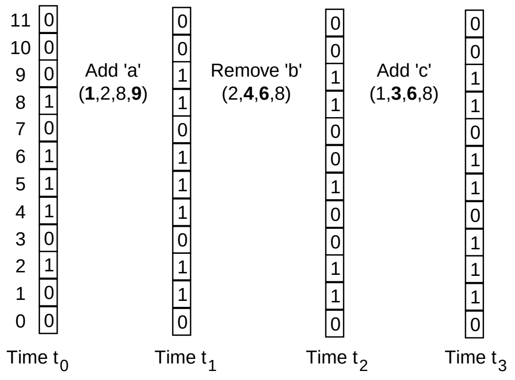
$$H_1 = h_{c,d}() | 0 < c < p, 0 \leq d < p$$

Here m is the size of the filter, which we calculated as the number of files stored at the node x the bits per file, which is part of the configuration for each experiment. Values for c and d were randomly generated by the Configurator at the beginning of each experiment. p was chosen to be a large prime number less than the maximum value of an integer on the machine we were using. Empirically we found that indices were well distributed over the size of the filter.

For bookkeeping, each node associated a saturation and a phase with its local filter. This information was not passed among nodes. The saturation kept track of the number of changes to a filter, and the phase noted the exact number of times a bit had been set to 1. With the phase, we were able to unset bits (and include removals in deltas).

4.2 Filter Deltas

In order to reduce the size of the messages being sent between nodes, we implemented a system of timestamps and filter deltas. Instead of only including new bits to "turn on" (or their indices), we send a bit string which is the size of the original filter with the bits the receiver needs to invert set to one. Because this array is sparse, it acts like a Compressed Bloom filter, and is highly compressible. In addition to keeping track of a filter to associate with each node, nodes must associate timestamps with each filter in their cache. They send this timestamp with every filter request and then the responder decides whether to send a new filter or a filter delta.



11
10
9
8
7
6
5
4
3
2
1
0
Add 'a' (**1**,2,8,**9**)
Remove 'b' (2,**4**,**6**,8)
Add 'c' (1,**3**,**6**,8)
Time t0
Time t1
Time t2
Time t3

Figure 4: Filter Deltas. At time t_0, the filter is created; at time t_1, file *a*, which hashes to (1,2,8,9) is added. Because bits 2 and 8 are already set, only bits 1 and 9 (in bold) are changed (although the phase at all four locations is updated). A time t_2, file *b* is removed; 2 and 8 is not switched off because their phases are greater than zero. At time t_3, file *c* is added. If a requester's timestamp is t_1, the counter would put -1s at indices 4 and 6, and then add one at locations 3 and 6, giving index 6 a net value of zero. The delta bit array sent back to the requester would then have bits 3 and 4 set. The requester would flip these bits, setting index 3 to "on" and index 4 to "off," giving it the correct current filter.

Obviously, if the requester's timestamp is earlier than the origin of the filter (i.e. t_0), the requester must be sent the entire filter.

4.3 Protocol

The protocol used to communicate among the nodes becomes significantly more complex as it move from the world where every node is a neighbor to the world of groups, representatives, and proxied queries. The protocol for a fully-connected system works as follows:

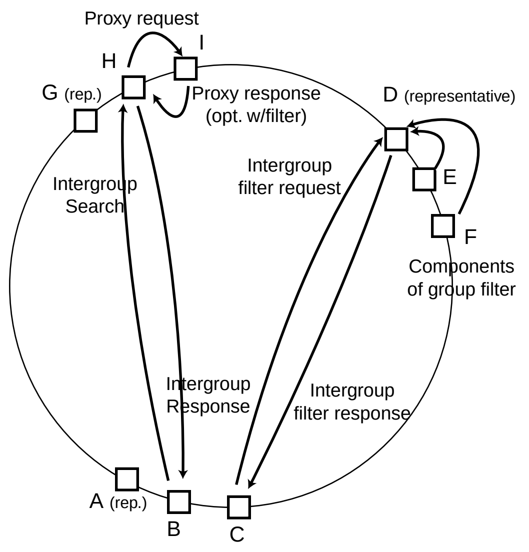
Because the responder keeps a list of which bits were turned on with a file add or off with a file removal, it can generate exactly which bits need to be set in the requester's filter, given the timestamp of the requester's current filter. Which bits to flip is determined by the following algorithm:

1. Create an empty integer array the size of the filter, initializing all slots to zero.
2. Each time the bit is set to one, increment the counter at that slot. Decrement when the bit is unset (e.g the removal at t_2).
3. Any bit that is non-zero, set this bit to one in the bit set sent to the requester of the filter.

VERIFY Verify that the receiver actually has a file and that the sender did not have a false bloom hit. Responds with either an ACK or a NACK.

ACK Node acknowledges that it has the file requested.

NACK Node says that it does not have the file requested. In our original implementation, NACKs would then always trigger a filter request from the receiver, because it assumed that its filter was out of date. To eliminate these two messages, the timestamp of the node's filter accompanies every VERIFY request and then a filter delta (or a whole filter) can piggyback on the NACK. ACKs also have the ability to port filters, and could do so if the timestamp showed the requester's copy of the filter was very out of date,



Proxy request
H
I
G (rep.)
Proxy response (opt. w/filter)
D (representative)
Intergroup filter request
E
Intergroup Search
F
Components of group filter
Intergroup Response
Intergroup filter response
A (rep.)
B
C

Figure 5: Grouping communication protocol. (A,B,C), (D,E,F), and (G,H,I) are groups. E and F send their group-sized components of group (D,E,F)'s intergroup filter. D, the representative of this group, *OR*s these filters with its own group-sized component and sends it to C, which has requested it. B is performing a search and its cache of intergroup filters has suggested that group (G,H,I) has the file it is looking for. It randomly chooses H to proxy this request to the rest of group (G,H,I). H looks at its more precise intragroup filters and at its locally shared files and determines that I might have the file B is looking for. I responds to H with a NACK, which then in turn responds to B with the same. B will then ask G, (G,H,I)'s representative for a new intergroup filter.

but this was not used. The Search object for this file is contacted and it initiates a new VERIFY request if there are more possible nodes to contact or it signals that the search has completed unsuccessfully.

FILTER REQUEST Node requests that the receiver sends it the receiver's filter and timestamp.

FILTER RESPONSE Node receives filter from a neighbor and adds it to its cache of filters, possibly by applying deltas.

Currently, the initial bootstrapping of network discovery is part of the configuration received from the Configurator, but the ability to discover the network existed in the protocol of an early implementation.

The grouping topology and communication is more complicated but based on the same protocol. The grouping topology, seen pictorially in Figure 5, consists of the same messages as in the fully-connected case, followed by a flag which further describes the action to take. These flags show whether the action is: (1) within the group (intragroup), (2) among groups (intergroup), (3) between a group and its representative (representative), (4) for an intergroup proxied search (proxy).

VERIFY Intragroup follows the same form as above, in the fully-connected protocol. Between groups, this initiates a proxy search, where a randomly chosen node in a group uses its filters to search for an extragroup node. A proxy verify message signals that the requester is performing a proxied search.

ACK Intragroup ACKs work as above. Intergroup ACKs signal the end of a successfully proxied search. Proxy ACKs come from within the same group and cause an intergroup ACK to be sent back to the query originator.

NACK Intragroup NACKs work as above. Intergroup NACKs signal a negative group proxied lookup and may initiate another intra- or intergroup VERIFY request if more filters match; otherwise there has been no match for the search. Proxy NACKs come from within the same group and initiate a lookup in the list of ongoing proxied searches; if more possible nodes from within the group are found, another proxy VERIFY message is sent, otherwise an intergroup NACK is sent to the originator. Filters can piggyback on both intragroup NACKs and proxied NACKs, as they are both always to members of the same group.

FILTER REQUEST Intragroup filter requests work as above. Intergroup requests are only directed to the group's representative, as only this node holds all of the composite filter components. Representative requests come from the group representative and instruct the receiver to respond with its intergroup filter component.

FILTER RESPONSE Intragroup this works as above, but more often these are piggybacked on intragroup and proxy NACKs. Intergroup filter responses send the extragroup node the logical *OR* of the constituents of this group; these only come from the representative.

5 Experiments

We examined network usage from two perspectives: (1) a more idealized, evenly distributed network where all nodes start off with the same number of files and add and remove files at the same rate and (2) an empirically derived model based on studies on the actual usage of file sharing networks like Gnutella [1]. (1) is actually not as idealized as it may seem because a collection of

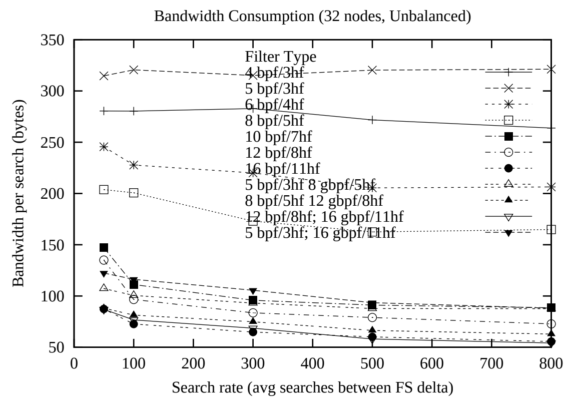


Figure 6: Bandwidth consumed per search in an “unbalanced” network, where very few nodes store almost all of the files.

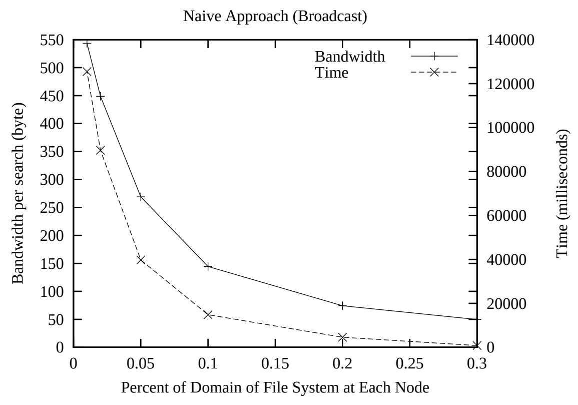


Figure 7: Bandwidth/search and time/search consumption when performing naive file queries

distributed file systems, like CFS and PAST, may follow a more balanced and controlled model of usage than the highly decentralized Gnutella. Adar and Huberman’s study on Gnutella usage shows how very few nodes are the sharers of the vast majority of files and that $> 70\%$ of nodes share none. They refer to this disparity as “free riding.” In a distributed file system, publishers may seek to load balance their own multiple publications and such a disparity may not materialize as these systems come into fruition. Most of our experiments follow the more idealized system of (1), although we do look at an unbalanced system in Figure 6. The unbalanced system follows the same behavior as seen in Section 5.5.

Because the tested system is fairly complex, with numerous variables to change, we mainly tried varying those which we postulated would have the largest affect. For example, we did not experiment with many different types of hash functions. Instead we varied the number of nodes, the $\frac{m}{n}$ rate, and the number of hash functions used, the rate of search (the amount of time the *Query* thread would sleep for between initiating new searches), whether deltas were used and whether they were compressed, and whether the filters themselves, when propagated in their entirety, were compressed.

5.1 Experimental Setup

All of the experiments presented have data collected on a system running with 32 nodes and each node generating a search request every second. The distributed file system has 4000 distinct files and each node generates 100 out of those 4000 files at system startup. All the nodes are threads running on the same machine and have a (port number, IP) pair that uniquely identifies them and enables the communication among them. At startup, each node receives a list of all the neighbors in the group. Also at startup each node contacts the *Configurator* process for a derived configuration file in which we specify different parameters such as run time, number of distinct files in the distributed file system, values for c and d, whether the node is a *Representative*, who are the members of the groups (if the experiment is using grouping), time between two consecutive searches, the type of search method used (Naive, regular Bloom filters, Compressed), etc. In the case when the configuration file specifies that Bloom filters are used, the thread generates a Bloom filter for the files in its file system, based on the parameters in its configuration file. Each process then starts its four threads, as described in Section 4, that send filter requests to other nodes and that begin generating file queries.

The experiments were run on machines with Linux 2.2.16 kernels, 800 Mhz Pentium III processors, and 1G RAM. The external compression process forked to perform delta compression and Compressed Bloom filters used /tmp on the root disk.

VERIFY, ACK, and NACK packet sizes were 20 bytes each. Filter message sizes depended on the bits per file of a given experiment. A NACK could also be large if a filter was piggybacked onto it.

5.2 Naive (Broadcast) Queries

In the naive approach, each node does a search by sequentially querying every node on the system until it gets a positive response. Thus the bandwidth consumed per search is dependent on the number of requests and responses sent per search — no filters exist to add bandwidth. The number of messages exchanged between the requestor and the rest of the nodes is dependent on the percent chance that the requested file is at the node being queried. Since every file from the distributed file system has an equal chance of being at the pinged node (in the test results for this experiment), the percent chance of a node having the file as the requestor searches for is the same as the ratio between the number of files at the node and the total number of files in the distributed file system. When a node has a large percentage of the files in the system, the chance of that node being able to send a positive response is higher. In our experiments we varied the ratio between the number of files at a node and the total number of distinct files in the system. The results are shown in Figure 7 on the left y-axis. As can be seen, the bandwidth per search grows almost exponentially as the number of files at the nodes decreases. In the case when a node contains 30% of the files in the file system, the false positive rate is 0.7 on average and the bandwidth is approximately 50 bytes per search. As the number of files at a node decreases, the false positive rate grows and, in the case when a node has 2% of the files, the false positive is about 98%.

Figure 7's right y-axis shows the time spent per search. Note that this time is a little bit higher than in reality since we do not account for searches that did not complete at the time when the tests ran for the specified period. The time spent also grows exponentially as the number of files at the nodes decreases. It starts with about 0.8 sec/search when a node has 30% of the files and goes up to 120 sec/search when a node has only 2% of the files. Note that our implementation of naive queries sequentially asks neighbors; i.e. it first waits for the neighbor's response before asking the next neighbor. There obviously could be a time improvement at a higher bandwidth cost if a node sent all requests in parallel to all neighbors. In this case, searching for a given file, the time would take just the round trip time to a node on the network, if the network could sustain this usage. However, the bandwidth expense will be n $\times$ (bandwidth for a request + a response) where n is the number of nodes on the network. The bandwidth usage per search will always be the same and will be equal to what the bandwidth per request is in the sequential naive case that we implemented with nodes having 2% of the files of the file system (note that in the case when a node has only 2% of the files we are likely to query all the nodes).

In conclusion, for file systems in which the nodes have 30 or more percent of the files in the system, sequential search will be better. In the case when a node has less than 10% of the files, sending simultaneous requests to everyone will work better.

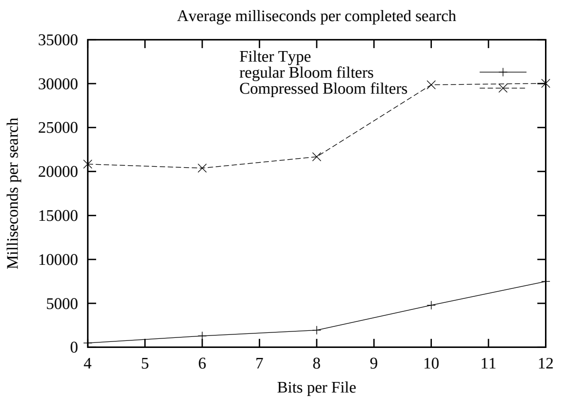


Figure 8: Time per Search

5.3 Search Time

In Figure 8, we compare the average amount of time required to complete a file search in our system for two search mechanisms: standard and Compressed Bloom filters. Grouping times, because they are much smaller are given in the following table:

Group Combination	Milliseconds per Search
5 bpf/3hf, 8 gbpf/5hf	35.4
8 bpf/5hf, 12 gbpf/8hf	39.4
12 bpf/8hf, 16 gbpf/11hf	182.1
5 bpf/3hf, 16 gbpf/11hf	62.4

The groups use non-compressed Bloom filters. The experiments were run for 15 minutes in a system with a rate of 800 searches per file change at a node.

Search time is defined as the elapsed time from the moment a query is submitted until the moment either the first positive acknowledgement is received or the last contacted neighbor replied negatively. Note that in the case of filters, our definition accounts for the amortized time required by updates and initial set-up phase, as well as the time necessary for hashing and sequential filter checking at the node that generated the query.

Time per search for all filter configurations increases with the number of bits per file used at the nodes. A higher number of hash functions and longer transmission times account for the almost proportional increase of time per search as a function of bits per file in the case of Bloom filters. Although the false positive rate drops, longer processing and transmission time for the larger filters compensate for the lower frequency of updates.

To compare processing time at nodes for regular and compressed filters we selected the parameters of the latter such that bandwidth consumed per search is approx-

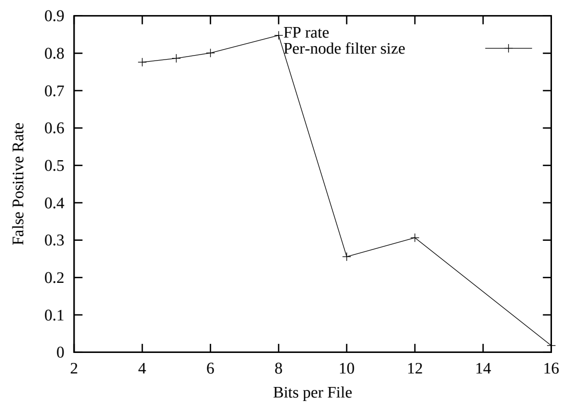


Figure 9: Observed false positive rates for differing bits per file. Rates have been averaged over five query/fs delta rates.

imately equal for the two filter types. Due to the slow compression/decompression mechanism, time per search in the case of compressed filters is about 10 times larger than for the corresponding standard filter with identical transmission size. Our experiments indicate that, at least for a small system like ours (32 nodes), the compression/decompression operations dominate processing time, such that time savings from faster hashing (smaller number of hash funtions) and smaller false positive rates are insignificant.

Time per search for the grouping design is more than 100 times smaller than the smallest time for fully-connected setup with regular Bloom filters. This is correlated with smaller bandwith consumption for groupings, so it is mainly due to smaller overall transmission time.

Similar to our observations on bandwidth consumption, the naive query protocol performs better than regular Bloom filters timewise when the percentage of total files in the system owned by any node is higher than 10%. Otherwise, time per search using the standard Bloom filter mechanism is smaller. Groups compete well with the naive protocol even when the percentage of files owned by nodes is fairly large. On average, time per search using any of the grouping parameters is less than 79 ms, while time per search for naive queries when nodes own 30% of all the files is about 839ms.

5.4 False Positive Rates

In Figure 9 we plot average achieved false positives rates against number of bits per file. The latter is computed as number of *NACK* messages (number of contacted nodes that responded negatively to a query) over the number of verify messages (total number of nodes contacted). In retrospect, we believe that we are not recording the false positive rate correctly because we are not recording the behavior on a per-filter basis, only on a per-node's cache basis. In other words, we are not keeping track of the total number of *NACK* messages generated by a particular filter and dividing by the total number of *VERIFY* messages this filter has generated. We are confident that the problem is one with measurement and not with implementation (and we are unable to extract the information to compute the rate in this different way from our current completed experiments).

Even with this proviso, the false positive rate achieved in our system does not entirely comply with the predicted minimum probability of a false positive, which decreases exponentially with the number of bits per file. For 4, 6 and 8 bits per file the system's false positive rate is particularly high having a value of about 80%. It then decreases drastically to about 5% for 10, 12 and 16 bits per file.

5.5 Bandwidth Consumption

As discussed in Section 5.2, the bandwidth consumption in the case of the naive querying protocol depends on the percentage of files in the system owned by each node. In contrast, percentage file ownership at nodes does not affect the performance of Bloom filters since each member of the system contains the Bloom filters of all other nodes, and therefore have equal information regarding the different files its neighbors possess. In Figure 10 we examine the variation of bandwidth consumption per search for standard Bloom filters as a function of the number of searches per file system change (i.e., a measure of the frequency of updates in the system) for the fully connected and grouping system designs. Bandwidth consumption is divided into bandwidth used by filter updates and verification messages (*VERIFY*, *ACK* and *NACK* messages).

For the fully-connected network set-up, we ran experiments with 4, 6, 8, 10 and 12 bits file and optimal number of hash functions, 3, 4, 6, 7 and 8 hash functions, respectively. For the grouping design, we experimented with the following combinations of bits per file and hash functions for the intergroup filters: (5, 3), (8, 5), (12, 8). With these we associate "more precise" combinations of bits per file and hash functions for the intragroup filters: (8,5), (12, 8), and (16, 11).

We observe that in the fully-connected system, there is a tradeoff between memory consumption at end nodes and network traffic. Network traffic for Bloom filters in the fully-connected set-up is highly correlated with the false positive rates. The false positive rate of our system remains high at about 80% for 4, 6 and 8 bits, and it drops steeply below 5% for 10, 12 and 16 bits per file. As a consequence, the combined average bandwidth per search for

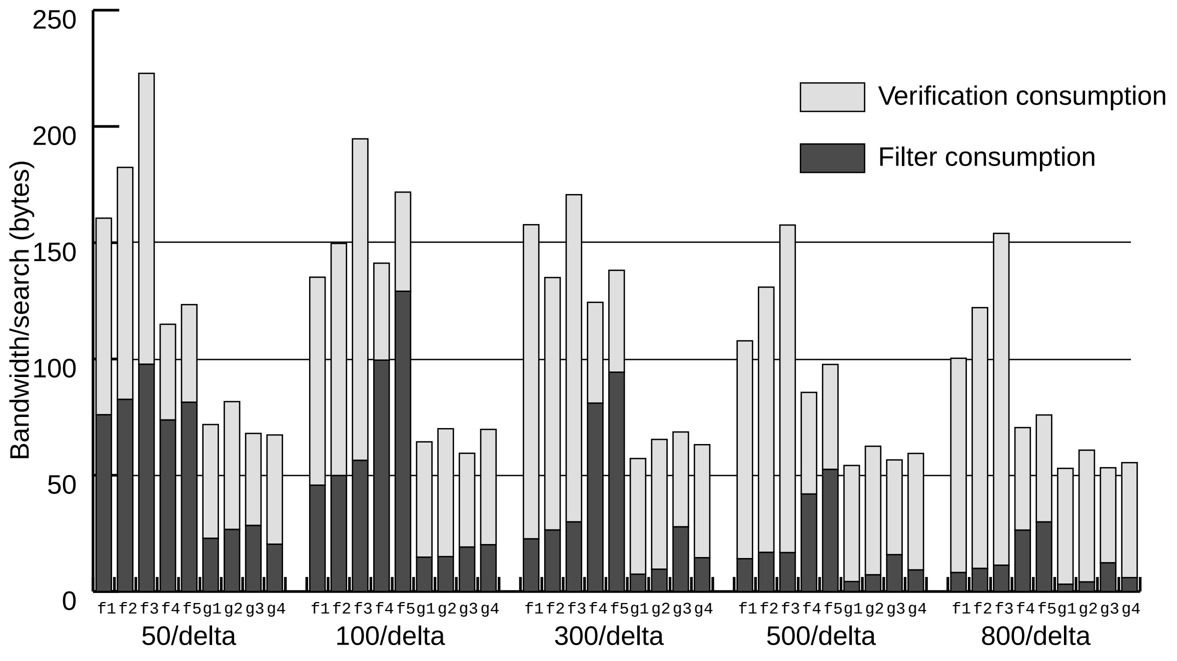


Figure 10: Bandwidth consumed at differing rates of searches per file system update (e.g. 50 searches on average per change in an average node's file system). Bandwidth is split into its two components, verification messages (*VERIFY*, *ACK*, *NACK*) and filter message (*FILTER REQUEST*, *FILTER RESPONSE*, *NACK WITH FILTER*, where the filter has been piggybacked onto the *NACK* message when there has been a file system change at the queried node). The notations on the x-axis correspond to the following bits per file / hash function combinations: f1 $\rightarrow$ 4 bits per file, 3 hash functions; f2 $\rightarrow$ 6 bpf, 3 hf; f3 $\rightarrow$ 8 bpf/5hf; f4 $\rightarrow$ 10 bpf/7hf; f5 $\rightarrow$ 12 bpf/8hf; g1 $\rightarrow$ 5 bpf/3hf; 8 gbpf/5hf; g2 $\rightarrow$ 8 bpf/5hf; 12 gbpf/8hf; g3 $\rightarrow$ 12 bpf/8hf; 16 gbpf/11hf; g4 $\rightarrow$ 5 bpf/3hf; 16 gbpf/11hf

4, 6 and 8 bits per file is 21% higher than the combined average bandwidth per search for 10 and 12 bits per file. Therefore, nodes can reduce network traffic by decreasing the false positive rate at the expense of higher memory requirements at the end nodes. In contrast, the grouping setup shows little variation with the false positive rate of the system. While the false positive rate varies from 75% to 28% in the three situations we looked at, the bandwidth consumption remains around an average of 63 bytes per search.

As predicted by our theoretical considerations, the bandwidth per search in the case of grouping is always significantly lower than bandwidth per search in the fully-connected setup, and decreases only slightly with the search rate per file change. In particular, the average bandwidth consumption for groupings is about 50% lower than the average bandwidth consumption for 10 and 12 bits per file in the fully-connected set-up.

In contrast, the average bandwidth per search in the fully-connected setup decreases as the number of searches per file change at nodes increases. This is expected since more searches per file change implies fewer updates per search, and therefore lower bandwidth consumption. This is confirmed by the fact that, on average, the ratio of filter to verification bandwidth consumption decreases as the search rate per file change grows.

Recall that in the case of naive queries bandwith consumption increases exponentially as the percentage of files owed by a node declines below 10% of the files in the system. Namely, bandwith increases from 50 bytes per search when nodes own 10% of the total files, to 150 bytes when they own 5%, and to 350 bytes when they own 1%. In contrast, the Bloom filter bandwidth consumption does not vary with the percentage of files owned by nodes, and ranges between an average 50 bytes per search for groupings and 175 bytes per search for Bloom filters with high false positive rates. Therefore, in a system where nodes own less than 10% of the total files, Bloom filters are a clear bandwidth saving search mechanisms.

5.6 Compressed Bloom Filters

We compare the bandwidth consumption of standard and Compressed Bloom filters for small and medium false positive rates in the system. To tune the compressed filters' parameters we picked the theoretical false positive rates for regular Bloom filters with 8 bits and 16 bits per file, 0.0216 and 0.00049, respectively. In practice, we obtain an average false positive rate of 0.270 and 0.00920.

For a given false positive rate, we run the system with the available combinations of bits per file at nodes and number of hash functions that yield a theoretical false positive rate closest to the desired rate and a theoretical number of transmitted bits per file below at least 90% of the bits per file ratio required by the optimal regular filters corresponding to that false positive rate. The following table shows the choices of bits per file at nodes, number of hash functions and expected number of transmitted bits per file for the small and medium false positive rates considered.

Bits per File	Hash func	Exp Trans	Exp FP
8	6	8	0.0216
9	3	5.36	0.0227
10	3	5.72	0.0174
13	2	5.32	0.0203
46	1	4.77	0.0215
16	11	11.09	0.00045
21	5	10.84	0.00042
26	4	10.65	0.00041
38	3	10.20	0.00043
93	2	9.57	0.00045

Figure 11: Compressed Bloom Filters: Expected Transmission bits per File

The experiments were run for 16.67 minutes in a system with 32 nodes, where each node generates 500 searches per file change. We noticed that due to the large compression/decompression time requirements, the initial set up period (i.e., the period between the time when the system is started until every node receives and decompresses the filters of its neighbors) of the system is much longer when compressed filters are used. In reality, the system would be run for a sufficiently long time such that the additional compression overhead is amortized across searches. Since in our experiments the system was run for a relatively short period of time, our analysis ignores the bandwidth consumed during the initial set-up to avoid distortion of the results.

Bits Per File	Band/srch	Nack w/filter	FP rate
8	54	172	0.221
9	59	187	0.333
10	60	187	0.346
13	58	219	0.219
46	50	178	0.232
16	41	303	0.0056
21	42	406	0.0074
26	42	365	0.0078
38	44	310	0.0159
93	42	285	0.0091

Figure 12: Compressed Bloom Filters: Bandwidth Consumption per Search. Bandwidth and Nack with filter sizes are in bytes.

From this table we note that, contrary to expectations, bandwidth per search is on average 4.47% and 3.12% higher than in the case of regular Bloom filters with 8 bits and 16 bits per file respectively under all parameter combinations. Several reasons explain our results. First, the size of the uncompressed Bloom filters is not sufficiently large to achieve optimal compressions with arithmetic encoding. Due to memory constraints, we were prohibited from simulating larger file systems, with more bits to compress. In all our experiments, end nodes own 100 files such that the size of the largest uncompressed filter in our experiments is 1163 bytes. To show that, we compute the size of a *NACK* with filter for each choice of parameters. Observe that for less than 93 bits per file, filters compress to more than the size of the regular Bloom filter. However, using 93 bits per filter compressed to 285 bytes, while the corresponding regular Bloom filter is 303 bytes long. In addition, we note that although the false positive rates for standard and compressed filters should be identical (the parameters of the compressed filters were chosen such that a given false positive rate is maintained), the false positive we obtain for Compressed Bloom filters is on average slightly higher than the matching rate for regular filters. We believe this result might be due to delayed updates caused by the lengthy compression/decompression process. In our implementation, filter compression requires a forked process and several input/output operations, which add significant overhead to the actual compression.

Our results suggest that Compressed Bloom filters would most probably improve bandwidth in large distributed files system where the number of files at nodes are significantly more numerous that 100.

6 Conclusion

Our initial plan was to derive some formula where a particular instance of a distributed file system using filtering to enhance filename queries could plug in the number of nodes it had and the rate of change of its constituent file systems versus the rate of queries, and out would come the right filter dimensions. We have found that the number of variables is large and significantly interdependent — initial experiments with fewer nodes showed different results than with 32 although patterns were clearly emerging. Even with this interdependence, we believe that our grouping construct provides a scalable alternative to naive searching and to hop-based schemes.

In the future, we would like to experiment on far more nodes and include an implementation of Sripanidkulchai's proposal [21], where domains with similar interests are grouped together. Although this was not tested, we postulate that this grouping scheme would achieve even better results if combined with his scheme, described in section 2.2.

We would also like to perform a more thorough analysis of Compressed Bloom filters, in particular when they are used with aggregation and deltas when the compressor itself is not a major bottleneck. In particular, we think the large, sparce constituents of the intergroup filters, sent to the group representatives, would compress well. Because the nodes are written as separate processes and get their configuration remotely, running them on many machines may not be very difficult. We would also like to analyze the actual false positive rates better; our current implementation does not keep per-filter statistics and these could be informative. We are confident in the underlying Bloom filter implementation, however, as we verified it with several separate experiments, including running it against a standard UNIX dictionary, and the results matched the theoretical expectations.

After a more thorough analysis of the tradeoffs in intragroup and intergroup filter size and when to propagate filters based on file system changes, we believe that Bloom filters and the network topology we have constructed will be ready for a large-scale implementation on top of an existing distributed file system, like CFS or PAST. To twist an old aphorism, users cannot find what they cannot see; we think this will let them see.

References

[1] Eytan Adar and Bernardo Huberman. Free riding on gnutella. *First Monday*, 5(10), October 2000.

[2] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[3] Ian Clarke. Freenet: A distributed anonymous information storage and retrieval system. `http://freenetproject.org/cgi-bin/twiki/view/Main/ICSI`, 2001.

[4] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, October 2001.

[5] Peter Druschel and Antony Rowstron. Past: A large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, October 2001.

[6] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 2000.

[7] Gnutella. Gnutella protocol specification v0.4. `http://www.clip2.com/GnutellaProtocol04.pdf`, 2001.

[8] John Howard, Michael Kazar, Sherri Menees, David Nichols, Mahadev Satyanarayanan, Robert Sidebotham, and Michael West. Scale and performance in distributed file system. *ACM Transactions of Computer Systems*, February 1988.

[9] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. In *Thirteenth ACM Symposium on Operating Systems Principles*, 1991.

[10] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM, November 2000.

[11] Lothar F. Mackert and Guy M. Lohman. R* optimizer validation and performance evaluation for local queries. In Carlo Zaniolo, editor, *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 28-30, 1986*, 1986.

[12] Michael Mitzenmacher. Compressed bloom filters. In *Twentieth ACM Symposium on Principles of Distributed Computing (PODC 2001)*, 2001.

[13] A. Moffat, R. Neal, and I.H. Witten. Arithmetic coding revisted. *ACM Transactions on Information Systems*, 16(3):256–294, 1998.

[14] James K. Mullin. A second look at bloom filters. *Communications of the ACM*, 26(8):570–571, 1983.

[15] Rasmus Pagh and Flemming Friche Rodler. Lossy dictionaries. In *Algorithms – ESA 2001*, volume 2161 of *LNCS*, pages 300–311. Springer, 2001.

[16] M. Ramakrishna. Practical performance of bloom filters and parallel free-text searching. *Communications of the ACM*, 32(10):1237–1239, October 1989.

[17] Matei Ripeanu. Peer-to-peer architecture case study: Gnutella network. In *Proceedings of International Conference on Peer-to-peer Computing*, August 2001.

[18] Jordan Ritter. Why gnutella can't scale. `http://www.darkridge.com/~jpr5/doc/gnutella.html`, 2001.

[19] R. Sandberg, D. Goldberg, S. Kleinman, D. Walsh, and B. Lyon. Design and implementation of the sun network filesystem. In *Proceedings of the Summer 1986 USENIX Technical Conference*, 1986.

[20] Kunwadee Sripanidkulchai. The popularity of gnutella queries and its implications on scalability. `http://www-2.cs.cmu.edu/~kunwadee/research/p2p/gnutella.html`, 2001.

[21] Kunwadee Sripanidkulchai, Bruce Maggs, and Hui Zhang. Enabling efficient content location and retrieval in peer-to-peer systems by exploiting locality in interests. `http://detache.cmcl.cs.cmu.edu/~kunwadee/research/papers/sigcommposter0%1.ps.gz`, 2001.