Panaché: A Scalable Distributed Index for Keyword Search

Tim Lu, Shan Sinha, Ajay Sudan
{timlu, ssinha, ajaytoo} @mit.edu

Abstract

The primary challenge in developing a peer-to-peer file sharing system is implementing an efficient keyword search mechanism. This paper presents Panaché, a distributed inverted index that scales well with the number of nodes in the network. Panaché addresses three critical needs for searching peer-to-peer file sharing systems—efficient use of bandwidth, relevant search results and accommodation for graceful node transience. To achieve these needs, Panaché aggregates popularity information and builds upon other peer-to-peer systems that distribute index information by keyword. Relying on a combination of Bloom filtering, query ordering, and truncated results based on popularity data, Panaché can be shown to use significantly less bandwidth than Gnutella using real-world estimates of network parameters, while retaining high quality search results. Simulation experiments demonstrate that Panaché may be viable for Internet deployment, although more comprehensive testing is needed. Panaché provides an exciting starting point for future development and optimization.

1. Introduction

As hard disks have become inexpensive and broadband Internet has become more widely available, Gnutella has quickly become one of the most popular peer-to-peer (P2P) file sharing overlay networks on the Internet. The most significant challenge to designing a P2P file sharing system (FSS) is providing a keyword-search mechanism that allows users to efficiently locate relevant documents. In this paper, we present Panaché, a distributed inverted index that utilizes popularity information to provide efficient keyword search capabilities for P2P file sharing systems.

Panaché is intended to be used for the storing and searching of text descriptions of documents by P2P FSS users. The actual mechanism by which documents are transferred between peers is left to be solved by an external system. Panaché remains independent of the actual file sharing mechanism. Thus, Panaché makes no claims regarding the integrity of retrieved files.

The assumptions about the network made by Panaché's design are consistent with those of systems like Gnutella and Napster. Documents, primarily multimedia files that do not contain hyperlinks between them, are scattered across the network on individual users' machines. Panaché does not have control of the files that are indexed but maintains pointers to locations of files in the network. In order to participate in the index, we expect users to download and install the Panaché client on their machines. Panaché's index is generated by users that explicitly publish their documents using the Panaché client.

Whereas Napster used a central index, we perform indexing using a decentralized model. Gnutella, on the other hand, does not maintain an index; it provides a search mechanism that consists of broadcasting queries, resulting in a breadth-first walk through the network of participating hosts [8]. Napster's approach is efficient but is vulnerable to legal and political scrutiny. Gnutella's keyword search model is distributed, but inefficient as the number of nodes grows, since the entire network must be searched to find all documents matching a given query.

The goal of Panaché is to provide an efficient distributed index for an Internet P2P FSS that scales well in the number of nodes. We claim that Panaché scales better than Gnutella.

Panaché distributes its index by partitioning along keywords as suggested by Reynolds and Vahdat [13]. The unique capability that Panaché provides beyond typical keyword partitioning is the use of popularity information. We define popularity in the context of an Internet P2P FSS as the number of hits generated by users that retrieve documents from the search results returned by Panaché. We define relevance by the assumption that documents that were chosen frequently by other users are more relevant. Inspired by Google, maintaining popularity information facilitates the generation of highly relevant search results [2]. In addition to increasing the relevance of search results, popularity information allows Panaché to introduce several possible optimizations to improve index efficiency beyond typical partitioning schemes.

Panaché's primary focus is on providing efficient use of bandwidth while maintaining reasonable query response times. There are four mechanisms that Panaché utilizes to perform efficient searches:

- Query ordering
- Bloom filtering of results
- Popularity information
- Truncated results

Panaché is optimized for indexing documents using a small set of unique keywords and is well suited for indexing files using descriptive words chosen by the

publisher or selected from document titles. We do not expect Panaché to perform well when indexing a set of documents based on all words in the text of the documents, as Google does [2].

A P2P distributed index must successfully handle users leaving and entering the network. In Gnutella, users typically remain on the network for approximately one hour [15]. User transience requires that index information be maintained as the network changes. Panaché implements a simple model for transience that reconfigures the index when nodes gracefully enter or exit the network. More sophisticated implementations are left for future work.

The rest of the paper is organized as follows. Section 2 examines other P2P implementations and distributed indexes. Sections 3 and 4 provide a description of our design and implementation. Section 5 describes our experimental tests and simulations. We outline future enhancements in Section 6 and summarize in Section 7.

2. Relevant Work

Our work builds on several sources of prior research. The underlying lookup mechanism of Panaché needs to locate the machine responsible for a certain keyword in sub-linear time. Several systems, such as Chord [16] and Pastry [14], provide a scalable distributed lookup mechanism. For n connected nodes, both Chord and Pastry require $O(\log n)$ routing hops and $O(\log n)$ routing table entries to deliver messages between nodes in steady state operation. As nodes join and leave the network in Chord, routing state is maintained with $O(\log^2 n)$ messages with high probability [16]. Panaché uses Chord to map keywords to servers, but other distributed lookup mechanisms, such as Pastry [14], Kademlia [11], and Tapestry [18], could have been used as well.

Several existing P2P networks have adopted differing architectures to service search requests. Gnutella [8] broadcasts queries to locate files that match particular keywords, with each node contacting all of its neighbors. The aggregate bandwidth required to support a query is costly. To limit message propagation, a time-to-live (TTL) is assigned to each query. However, TTLs may result in suboptimal responses, since a document may reside on a node that is never contacted. Yang and Garcia-Molina [17] propose several methods for minimizing the bandwidth consumed during search in Gnutella-like systems, such as iterative deepening, directed breadth-first search, and local indexes, where nodes store local information about files on neighboring nodes. KaZaa [9] promotes machines with wider bandwidth to Supernodes and makes Supernodes responsible for indexing files stored by nodes around them. Freenet [5] has no explicit keyword search mechanism for files. Napster [12] relied on centralized servers to answer search queries.

Panaché builds upon a research system proposed by Reynolds and Vahdat [13] by maintaining statistics about popular documents and hosts. Their system uses Bloom filters to compactly represent set membership and joins the indexes matching the query keywords. KSS [7] assumes that network bandwidth is more precious than storage space in P2P networks. Rather than indexing single keywords, KSS builds a distributed inverted index keyed on combinations of multiple keywords. Thus, KSS avoids the overhead of computing joins but requires an insertion time that grows exponentially with the set size of the keyword combinations used to build the inverted index [7]. A novel approach by DINX [6] spreads popular documents over more nodes in a Chord ring and queries random nodes to find documents. DINX performs well for locating popular documents, but requires $O(n)$ time to find unpopular niche documents since all machines in the system must be contacted.

The aggregation of popularity data in our system is inspired by the ranking system used by Google [2]. In Google, a quality ranking for web documents is calculated based on the number of other pages that link to the given page and several other metrics.

3. Design Overview

Panaché is layered on top of Chord, which is an efficient distributed lookup protocol for a given key. Chord contains a set of nodes, each with a unique identifier, a ChordID. It provides a single function, *successor(key)*, which returns the successor node of *key*. The successor is defined as the first node whose ChordID is equal to or follows *key* in the identifier space, illustrated in Figure 1 [16]. The index is partitioned by routing queries for a given keyword to a Chord node. Keywords are mapped to the ChordID space by using their 160-bit SHA-1 hash.

Panaché's keyword partition scheme is based on Reynolds and Vahdat [13]. Chord is a natural choice for the lookup mechanism since it provides a simple interface that scales logarithmically in the number of nodes, which meets our goal of scalability.

Each Chord node in Panaché must run the Panaché client, containing two components, an index server and a query agent. Index servers are responsible for maintaining their indexes and satisfying requests from query agents. Panaché utilizes the libasync library that comes with the SFS software package developed by PDOS at MIT. The query agent is responsible for making requests to index servers. The index server and query agent primarily utilize libasync's RPC package for asynchronous communication. Since the RPC package uses UDP, messages that require the transfer

of large blocks of data occur over an auxiliary TCP port available on all Panaché clients. Using TCP for large messages avoids the retransmission penalty for lost packets that contain large amounts of data.

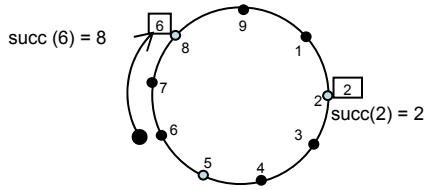

Figure 1. Chord identifies the responsible node for a keyword using the successor function. The *keyword* for which *hash*(*keyword*) = 6 maps to the node with ChordID 8, since *successor*(6) = 8. S*uccessor*(2) returns the node with ChordID 2 in the network.

4. Implementation

4.1 Data Structures

Each index server stores two basic data structures, **Documents** and **Hosts**. Document**s** contain a name, a counter indicating the hit count, and a hash of the document's content (to identify identical documents). Hosts contain a document name, a location (URL), and a counter indicating the number of times the file has been downloaded from this location.

Panaché maintains two hashtables, one mapping keywords to documents, used for query processing, and the other mapping documents to hosts, used for document retrieval. The first table, called the **DocTable**, maintains popularity information for documents, independent of their locations in the network. Since multiple keywords may hash to the same bucket, each bucket contains a reference to a series of linked keywordLists. The keywordList contains a binary search tree (BST), **docBST**, of all documents with the specified keyword, as shown in Figure 2. Documents are inserted into the BST based on their popularity as determined by their hit count.

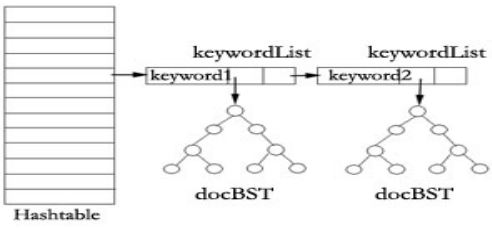

Figure 2. The DocTable hashtable used in Panaché contains keywordLists that include docBSTs of all documents corresponding to a specific keyword.

The second hashtable, the **HostTable**, maps documents to hosts. It may maintain quality rank information on publishers, although this was not implemented. The second table also allows the downstream FSS to distribute the load of document retrieval among multiple locations, which allows the possibility of doing parallel retrieval. The HostTable implementation is identical to the DocTable, except that it contains document/host pairs, which are keyed on the document name.

Hashtable put and get operations run in *O(*1*)* time. BST insertion and accesses take *O(lg n)* time on average. Document selection on docBSTs is done using *select(k)*, which returns the document with the k-th highest popularity. Selection uses a dynamic order statistics algorithm (`OS-SELECT`) described in CLRS with an expected *O(lg n)* running time [4]. This can be improved to a worst-case running time of *O(lg n)* by using Red-Black Trees.

4.2 Document Publishing

Documents are explicitly published by users in Panaché. Since Panaché was initially built to index multimedia files, a crawler was not considered as part of the design. Multimedia files typically do not contain hyperlinks that can be crawled. However, Panaché features a command interface for which a crawler interface could be built.

The first time a user enters Panaché, a bulk insert must be performed on all documents the user wishes to publish. To index a document, non-common words are extracted from the title. These words and optionally other user-specified words are the document keywords used to identify the index servers responsible for the document. The query agent sends an add document RPC request for each document to the appropriate servers, inserting the document and host information into its index. Each request contains the name and hash of the document. Currently, only the owner of the document is enabled to publish, since the IP address of the message sender is used as the URL. A future extension to Panaché may include submitting a generic URL or a URI in the document data.

An initial bulk insert may require several thousands of RPC messages. However, in Section 5.1, we demonstrate this may be acceptably fast.

4.3 Querying

Multiple keyword searches are conducted by performing a database join on the indexes of the servers responsible for each keyword in the query. This is done by forwarding the query through a chain of servers that hold the index for each keyword. Each server successively intersects its index with the previous results and passes along the Bloom filter of its results, as described in Section 4.3.2 and illustrated in Figure 3. The process continues until the last server constructs the final answer and returns the resulting data set to the user. Empty intersections are reported back to the user immediately.

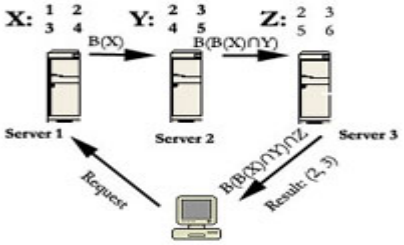

Figure 3. Query processing chain demonstrating the forwarding of Bloom filters B(·) to other servers and returning the final result to the client.

While large joins may be inefficient since large data sets are transported between servers, typical queries only contain 2-4 keywords [13]. Thus, the bandwidth needed during the search process is limited.

4.3.1 Ordering of Queries

To minimize the amount of data that must be transmitted between servers, the query agent sorts the keywords composing its initial query in ascending order of each keyword's index size. The final result can be no larger than the smallest index for a keyword in the query. The query agent obtains the data needed to order the results by contacting all of the relevant servers in parallel.

4.3.2 Bloom Filters

Bloom filters reduce the amount of forwarded data by a constant factor. Bloom filters compactly represent set membership in an approach outlined by Reynolds and Vahdat [13].

A Bloom filter of a set $S = \{x_1, x_2, \ldots, x_n\}$ is implemented as a bit array of size m with all bits initially set to 0 [3]. Each element x_i is hashed by hash functions $h_1, h_2, \ldots, h_k$, each of which map into the range $\{0, \ldots, m-1\}$. Each bit corresponding to location $h_i(x_i)$ is set to 1. To test membership of an element y in a Bloom filter, all $h_i(y)$ hashes are calculated and the corresponding bits examined. If any one of the bits is 0, then the element y is not in the set. If all of the bits are 1, then the element y either belongs to the set S or not; the latter case is a false positive. False positives emerge since many elements may hash to the same k bits. The process is shown in Figure 4.

To avoid computing k hashes per element, Panaché generates a SHA-1 hash of each document name. Each $\log_2 m$ bits of the hash indexes into the Bloom filter.

In order to minimize the number of false positives, the optimal number of hash functions k should be set to be $k = ln\ 2 * (m/n)$, where m is the size of the Bloom filter and n is the number of documents in the set to be represented [13]. This value of k yields a false positive rate of $(0.6185)^{m/n}$ [3]. The size of the Bloom filter can be optimized at each step in a query-chain; higher values yield lower false positive rates but also less compression. In this implementation, the ratio m/n was set to 15 to give a false positive rate < 0.01% and a compression ratio of > 16:1, given that each document record may be thirty bytes or longer.

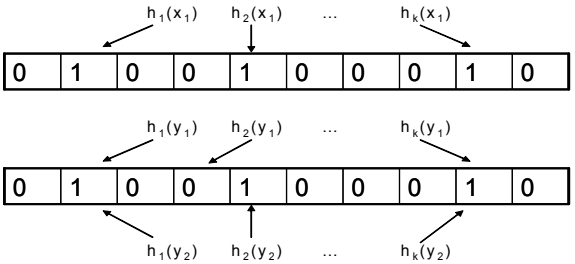

Figure 4. x_1 is inserted into the Bloom filter by setting all $h_i(x_1)$ bits to 1. Since $h_2(y_1)$ is 0, y_1 is not in the set. All of y_2's bits match, so it may be in the set.

Thus, servers transmit Bloom filters of results that represent document records. Upon receiving a Bloom filter, a server intersects it with the appropriate keyword index and constructs a new Bloom filter to be transmitted. The final results returned to the user are the actual documents. The query process may contain false positives. We do not believe that this is detrimental to user experience. First, the rate of false positives can be adjusted according to user preference. Second, users can filter out the small number of false positives manually. Finally, keyword search for a P2P FSS typically does not require perfect accuracy.

Reynolds and Vahdat [13] propose removing false positives by returning the final data set through the chain of servers again. Panaché could be modified to include this protocol as a user-selectable option, at the expense of additional latency and bandwidth.

4.3.3 Popularity

Popularity data helps to determine the most relevant results and enables optimizations for popular documents and queries. Popularity information is collected by counting the number of hits (hit count) that a given document or host has received. The user's query agent must notify the responsible servers when a document is selected by a user. Only the index servers that provided the results to a user's request are updated, meaning that other servers containing the same document may not receive the new information. The hit count measures a document's popularity for a given keyword. The same document accessed often by one keyword but not another should have a high hit count for the former keyword and not the latter when determining the relevant results to a given query. Thus, a global popularity ranking is probably unnecessary.

Results are returned based on their popularity ranking using the dynamic order statistics described in

Section 4.1. Section 4.3.4 describes how truncated queries use popularity to determine what documents to return to the query agent first.

One difficulty with using popularity data is how to determine popularity order when hit counts are initially low and no clear ranking exists. Since users are likely to select the first few results they receive, the first documents or hosts returned could become popular by default, despite their actual quality. As a potential solution, servers may periodically randomize all documents that have a hit count below a given threshold. When *select()* is called for documents with a hit count that is below this threshold, the results returned would be randomized. Thus, documents that have adequate popularity data above the threshold will be returned in order but those below will be randomized to avoid popularity inflation.

4.3.4 Truncated Queries

Though Bloom filters provide compression of data sets, query time scales with the number of keywords in a query and the number of matching documents to the query, as described in Section 4.6. Since users typically only view a limited number of results at a time, we provide the ability to obtain a constant number of matches at a time. Using popularity information, Panaché ranks the results during query processing to return the most relevant documents.

The first truncated query for t documents returns the top t results. If unsatisfied, a user can request the next page of t documents and so on, similar to the *next* capability on web search engines.

The query-chain is extended to support truncated results as follows. The first server forwards a Bloom filter for $2t$ documents that match the first keyword. The next servers check their indexes for matches and forward new Bloom filters along. If any server in the query-chain finds that less than t documents match, that server notifies the first server. The first server then restarts by sending a Bloom filter for twice as many documents as the prior iteration. Currently, Panaché does not save the current results for the restart; this will be fixed in the future to further reduce bandwidth overhead. If faster convergence is desired, instead of sending twice as many documents when restarting, a larger factor could be used so more documents are sent.

Breaking the query-chain process into chunks generates worst-case performance when the t documents to be returned are the lowest ranked ones on the first server. In this case, Panaché must transmit c times the number of documents in the first index to find those t documents, where c is the number of keywords in the query. On average, though, only $O(c)$ data needs to be transferred.

Truncated queries are similar to the incremental results described in [13]. They are somewhat similar to the TTL setting for Gnutella queries and iterative deepening in [17]. However, Panaché is more useful since it generates relevant documents in order instead of just returning the first set of documents found.

4.4 Transience

A distributed index for a P2P FSS must accommodate nodes that enter and exit the network. We implemented a simple transience model, primarily due to time constraints, that handles the graceful entry and exit of hosts. Handling node transience consists of transferring indexes between index servers.

An exiting host serializes its entire DocTable and HostTable and transmits this via TCP to its successor. After successful transmission, the host leaves. The receiving host deserializes the incoming data and inserts the keyword/document and document/host pairs into its own DocTable and HostTable, respectively.

To enter the Chord network, a node must, through an external mechanism, obtain the address of another node in the network. An entering host must then inherit a portion of the index from its successor. The new node notifies its successor to identify all keywords for which the successor is no longer responsible. The corresponding indices are transferred to the new node. Since Chord utilizes a ring structure for its key space, determining the keywords to transfer is simply a matter of hashing each keyword held by the successor and comparing it to the ChordIDs of the new node and its successor. The relevant entries from the DocTable and HostTable are serialized and sent to the new host via TCP. After successful transmission, the entries are deleted from the new host's successor.

Index transfers in our implementation present several challenges. For instance, after a new node enters the system and acquires its part of the index, it is unclear whether the transferred portion of the index can be deactivated from the original source, since there is latency in the settling time associated with Chord [16]. Settling time permits a situation in which two nodes simultaneously enter the network and temporarily resolve to the same successor. In our naïve implementation, once Chord settles, queries will continue to be resolved correctly, but this leads to unused, replicated data across several nodes.

Moreover, there are different policies associated with deactivating index entries. Suppose a node enters and inserts a new set of documents. If the node leaves and never returns, then the documents indexed should be expired by some mechanism.

Another challenge to index transfers is that they may be lengthy transactions, since a large number of index entries may have to be transmitted. For users

with modems, this may be a non-trivial wait time. Compression may help, but certainly there is an opportunity to develop a more sophisticated system.

We estimate that servers will index less than 10,000 documents in total [15], corresponding to approximately 400Kb. For a user connected by modem, the transfer may be long. However, the majority of indexes should be much smaller [15].

The current implementation of Panaché addresses none of the described challenges. Handling transience in a distributed index is closely tied to the issue of index reliability. We plan to address transience and reliability more completely in a future version.

4.5 Expected Performance

Panaché is expected to perform well for single keyword queries since identifying the correct index server requires *O(log n)* messages by Chord's lookup service. For multiple keyword queries, servers must perform joins of their respective data sets, yielding *O(k*m)* performance, where *k* is the number of keywords and *m* is the number of matches. For queries with few keywords or a low number of matching documents, query ordering and Bloom filters should yield good performance. For larger numbers of matching documents, truncated results can reduce the amount of data that needs to be transmitted over the network. Popularity information is used to ensure that the most relevant documents are returned. Nonetheless, Panaché may not perform optimally for queries composed of many keywords or for many matching documents since even with truncated results, a large number of documents must be ranked to return relevant responses.

According to Reynolds and Vahdat [13], about 28.5% of queries to popular search services were for a single keyword while 67.1% were for 2 to 5 keywords. Under the assumption that there are several tens of thousands of unique keywords [13] and a few million documents stored in a system like Gnutella [1], we expect that most keyword indexes will contain at most a few hundred entries. Thus, Panaché should yield satisfactory performance for the majority of queries. We acknowledge that documents are likely to follow a Zipf distribution [13], leaving some indexes particularly large. Several potential optimizations described in Section 6 address this problem.

4.6 Comparison with Gnutella

We claim that Panaché will outperform Gnutella in bandwidth utilization. Gnutella uses bandwidth inefficiently due to its broadcasting of queries. Every node must be searched to find all matching documents. Panaché utilizes a query routing protocol, along with some overhead to maintain index information, to eliminate the cost of searching the entire network. Thus, searches can be exhaustive without incurring the same penalties as Gnutella. However, Panaché requires overhead to maintain indices when nodes enter and exit, whereas Gnutella nodes may enter and exit freely.

We present a mathematical model based on real data from Gnutella to validate our claims of efficient bandwidth use. Let us assume the following parameters:

Time online	60 mins
# of keywords/query	*k*
Bytes per keyword	*b*
Queries per 60 minutes	*Q*
# of matching docs/query	*m*
Size of index entry in bytes	*s*
# of entries/host	*D*
# of nodes in the system	*N*
Gnutella query size in bytes	*R*
Average Bloom filter reduction	*B*

Table 1. Assumptions about the P2P environment.

Based on statistics from [8], [13] and [15], the following values seem appropriate : $k = 3$, $b = 5$ bytes, $N \sim 30000$ nodes, $R = 23$ bytes + $k * b$, $s = 100$, and $B = 0.1$. To compare the efficiency of the two networks, we will analyze the cost of carrying a node. We will focus on a time window of 60 minutes [15].

The cost of carrying a Gnutella node for one hour may be given by the following equation:

$$(1)\ C_G(n) = C_g(Q) + C(Q * m)$$

where $C_g(Q)$ is the cost of Gnutella queries from node *n* and $C(Q*m)$ is the cost of returned results. Thus,

$$(2)\ C_G(n) = QNR + Qms$$

Gnutella employs a TTL, which limits this value. However, we ignore the TTL since using the TTL results in the loss of complete network coverage. Furthermore, Gnutella carries a relatively high percentage of users that do not share any files, called freeloaders [1]. Freeloaders are essentially query forwarders and will exaggerate the loss of network coverage using TTLs.

The cost of carrying a node in Panaché may be given as follows, ignoring Chord RPCs since they are minimal compared to Panaché's traffic:

$$(3)\ C_P(n) = C_T(n) + C_p(Q) + C(Q * m)$$

where $C_T(n)$ is the transience cost, $C_p(Q)$ is the cost of Panaché queries, and $C(Q*m)$ is the cost of returned results. Equation (2) becomes:

$$(4)\ C_p(n) = 2Ds + QkmsB + Qms$$

Thus, $C_P(n) << C_g(n)$ if the following is true:

$$(5)\ \frac{2 * D * s}{Q * R} + \frac{k * m * s * B}{R} << N$$

Using reasonable values, `s = 100` and `Q = 5`:

$$1.05 * D + 0.789 * m << N \quad (6)$$

The most popular P2P FSS, KaZaa, maintains approximately 700 million documents for 3.78 million users [9]. Thus, a rough estimate for `D` is 185. Even in the context of the Gnutella network, where `N` = 30,000 and approximately 3 million documents are shared, `D` = 1000 [1]. Therefore, we expect that Panaché will not approach the bandwidth required by Gnutella's queries until `m` begins to exceed 30,000. As shown by Equation (5), Panaché's performance is expected to degrade with increasing `k`, the number of keywords per query. Although our system is best for indexing documents with a few keywords, research indicates that most queries are less than 5 keywords [13].

The optimizations introduced in Panaché explicitly address the issue of limiting the number of matching documents, `m`, by truncating results. Using popularity scoring, we are able to do so without significantly sacrificing search quality.

5. Experimental Results

Our experimentation involved simulating performance using the three 6.824 lab machines. As a result, our results were subject to noise. Due to resource constraints, we were unable to develop a heterogeneous environment for performance evaluations. In this section, we describe the data we collected as indicators of Panaché's performance.

5.1 Indexing

The time required for indexing documents over eight servers is shown in Table 2. We estimate that it should take 10 to 15 seconds to insert 1000 documents into the system with a 56Kbps modem connection.

10 Docs	100 Docs	1000 Docs	10k Docs
0.11 sec	0.12 sec	1.478 sec	20.32 sec

Table 2. Time needed for indexing new documents.

5.2 Querying

5.2.1 False Positives from Bloom Filters

We tested the false positive rate generated by Bloom filters using two keywords to populate the system with 100 matching documents. For the second keyword, we added a varying number of documents that would not match a query on the two keywords. Since our implementation requires Bloom filter sizes, m, to be powers of 2, m was 2048. With n equal to 100, this yields an m/n ratio of 20.48 and an optimal false positive rate of $(0.6185)^{20.48} = 0.0053\%$ [3].

As shown in Table 3, the average false positive rate was 0.013%. The actual rate was greater than expected since the number of hashes used was restricted to integer values in our implementation and could not be exactly optimal. We believe that the observed false positive rate is acceptable in a P2P search system.

# Non-Matching Documents	Ave # False Positives	False Positive Rate
10000	1.6	0.016%
20000	2.4	0.012%
30000	3.6	0.012%

Table 3. Average number of false positives and false positive rates for 10000, 20000, and 30000 non-matching documents for two keyword queries.

5.2.2 Query Time vs. System Documents

To determine the affect of the number of indexed system documents per server on query time, we populated 3 and 10 servers with varying numbers of total system documents and 100 matching documents and performed single and two keyword queries. System documents refer to non-matching documents for the test query. As shown in Figure 5, query times for a given number of total documents per server is roughly constant at several tens of milliseconds.

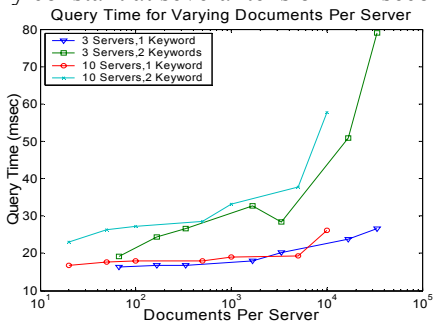

Figure 5. Query time for varying total system documents per server. Data was taken for 3 servers and 10 servers with single and two keyword queries and 100 matching documents.

Although query times degrade with higher index entries per server probably due to resource competition with non-matching documents, they typically remain under 100 ms for below 10^5 entries. Since most clients are expected to index less than 10,000 documents [15], the majority of index servers in Panaché will operate under 10,000 documents and will exhibit good performance. The minority that has more documents will still respond to queries with acceptable latencies.

5.2.3 Query Time vs. # of Query Keywords

As mentioned in Section 4.3.4, query time should scale with the number of query keywords. We measured a linear dependence of query time on the number of keywords. For 100 matching documents and 10 servers, single keyword queries took about 20 ms while 4 keyword queries took about 115 ms. Thus, performance will degrade for queries with many keywords, but queries typically contain less than five

keywords [13]. The graph showing this trend is omitted due to space constraints.

5.2.4 Truncated Queries

The benefits of truncated queries were measured by inserting increasing numbers of matching documents over 10 index servers and performing truncation queries with varying sizes. Panaché yields up to 200 results in less than 800 ms, for two keyword queries and 30,000 matching documents. As the number of matching documents grows, the truncated query time increases since more documents must be ranked and more passes through the query-chain must occur. The slight increase in query time is shown in Figure 6 for the two keyword case. Alternatively, each index could be periodically sorted by hit count so document ranks would not have to be calculated on the fly.

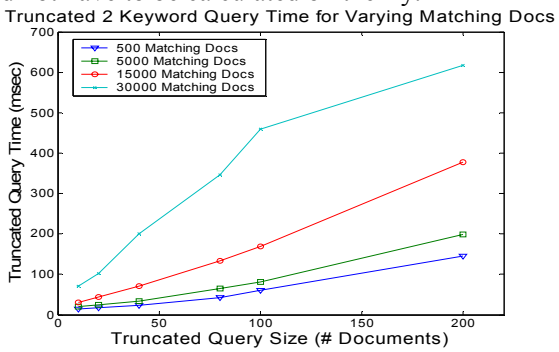

Figure 6. Truncated query time for varying numbers of matching documents and varying truncated query result sizes.

5.2.5 Query Time vs. # of Index Servers

Query time should increase with more index servers because more Chord messages have to be sent to locate responsible servers. Figure 7 shows that an order of magnitude increase in the number of index servers resulted in a minimal increase in query time.

Based on the trends shown here and Chord's fundamental sub-linear lookup operation [1], we believe that query times will not degrade drastically with many index servers.

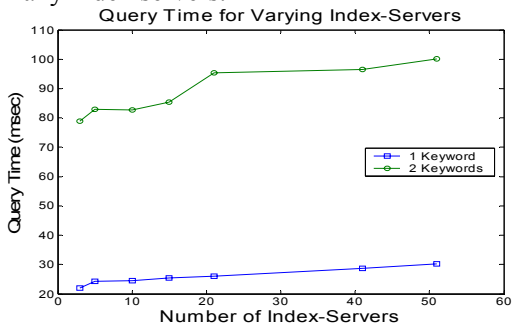

Figure 7. Query time for varying number of index servers in the system for 100 matching documents and single and two keyword queries.

5.2.6 Index Server Performance

Raw index server performance was determined by the number of queries per minute that could be satisfied within a reasonable amount of time. Three trials were conducted for varying index sizes.

We populated the server with a random test data set. A random number of keywords were generated, each with a maximum of 100 random documents out of a 1 million document set. Therefore, a random number of documents matched multiple keywords. We measured the response time for each query in a stream of queries. Each query was randomly made to be one to four keywords long. The keywords were selected from those used to populate the system, under the assumption that users typically know which keywords should be used in the search. Response times were averaged after streaming queries for 2 minutes.

Figure 8 demonstrates that the server successfully processed nearly 400 queries/min with a 50,000 document index. We believe these results are underestimates. First, activity from other users may have affected performance. Second, our implementation prevented us from writing a standalone client. Thus, our client ran on the same machine as the server, leading to resource contention.

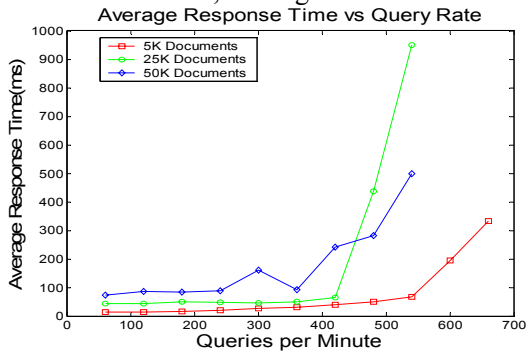

Figure 8. Raw index server performance measured against an input stream of random queries against a random distribution of documents.

Gnutella nodes must process approximately 45 queries per sec [10]. However, this is an inaccurate expectation for Panaché. First, Panaché queries are not as lightweight as Gnutella queries, since they may contain results that are to be intersected. Second, Gnutella broadcasts its queries, whereas queries in Panaché are routed. Routing reduces the number of queries that a particular node should process.

5.2.7 Bandwidth vs. Matched Documents

Bandwidth was measured by recording the total number of bytes that were sent between machines while processing a query. The protocol was modified to carry a cumulative byte count in each forwarded query. Figure 9 illustrates the amount of bandwidth

utilized during query processing. As expected, a linear growth in the number of documents is observed. Nonetheless, the crucial point of Figure 9 is that the bandwidth required for 32 index servers has barely increased beyond that needed for eight index servers.

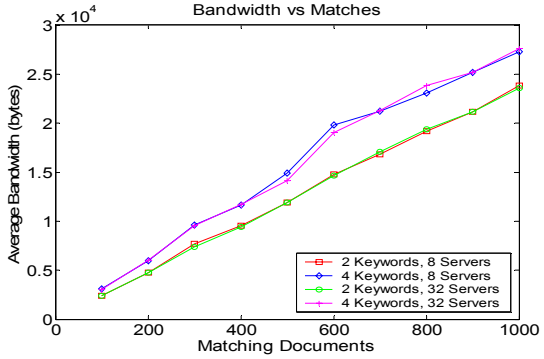

Figure 9. Bandwidth used during query processing. Systems with 8 and 32 index servers were measured.

5.3 Transience

To gauge the effects of transience, we were interested in the amount of bandwidth consumed and the time needed for host entry and exit.

Since the number of documents transmitted is fairly identical on an entry or an exit, we measured the number of bytes needed for an exit. The measurements do not include the underlying Chord messages used to lookup successors. The amount of data generated during an exit is plotted in Figure 10. On average, each document that must be transferred generated 38.8 bytes of data. Transmitting the index information for nearly 7000 documents resulted in 270Kb of data being transferred and took less than one second.

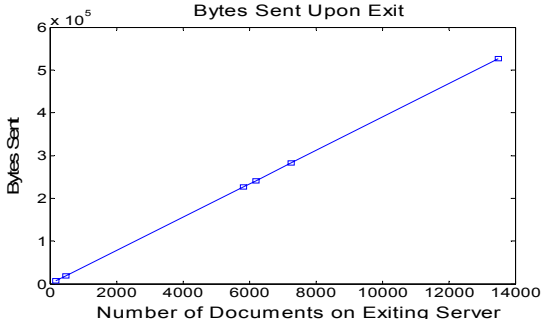

Figure 10. Bytes sent on exit versus the number of documents on the exiting server.

The transfer time needed during a transient event was measured as well. We populated one host with a variable number of documents. 20% of these documents were transferred to an entering host. The mean time for transmitting one document's indexing information was 8.5 µs. Transmitting 2000 documents took 170 ms; 5000 documents took 429.5 ms.

As described in Section 4.5, each host should index a few hundred documents on average. Even with a 56Kbps connection, transmitting index information for 100 documents takes just a few seconds.

6. Future Enhancements

There are several optimizations and enhancements that can be made to Panaché.

Query joins could be short-circuited. Each document entry could contain a list of all the keywords used to index the document. The first server to receive a query could filter the resulting set and return only the documents that contain all the keywords.

To further minimize network bandwidth consumed, compressed Bloom filters could be used [3].

Our current transience model does not handle staleness when a host leaves the network. A host's departure results in the loss of an index server and removes all the documents it is sharing. To prevent staleness, the host's departure would need to be broadcast to all hosts so that they can update their indexes and remove references to the departing server. This is inefficient as a significant fraction of hosts in the network may have references to the departing host. We realize this is a limitation of Panaché and propose two solutions to address the invalidation problem.

The first solution is to assign an expiration date to all documents when they are indexed. The date is delayed every time the file is accessed. In the absence of any notification from the document owner or in the absence of any accesses, the document is invalidated and removed from the index. The second solution is to have the client notify all index servers which processed the query that a result they returned is stale and should be removed. The two solutions could be combined to minimize staleness even further.

Currently, Panaché only supports the graceful exit of hosts. By replicating data on multiple hosts, Panaché could support network failures and unexpected disconnections from the network. Replicated data could restore the indexes on the remaining servers. Large or popular indices could be replicated more often to provide load balancing.

The result set produced for a query is currently returned to the client via an RPC call over UDP. Since the result set may be quite large, UDP is not well-suited to this task. Instead, the result set should be sent via TCP to be more robust to network errors.

Based on Panaché's popularity metric, an optimization based on KSS [7] could be implemented. If a particular two keyword query was detected to be extremely popular, the servers responsible for the keywords could insert a new index into the system corresponding to the popular query. The old servers could notify query agents for query ordering purposes.

To improve the response of Panaché to repeated queries, caching can be implemented on index servers.

Since query ordering is performed, the last server in the query-chain that responded to the same query previously can directly return the answer to the agent. This can also mitigate the cost of querying against extremely large indices.

To handle the varying amount of resources available to nodes in a heterogeneous P2P system such as Panaché, virtual nodes may be implemented. Physical nodes with greater system resources could be responsible for more ChordIDs and thus, documents.

Since we implemented our keyword search system separately from any particular P2P storage system, it is possible that Panaché could be used to do keyword search for Gnutella. This would require Gnutella to make a few modifications to insert documents in the index and to update the index based on popularity.

Our system design does not preclude the use of crawlers to build and update our index. This may be desirable for index construction and for certain applications that require data freshness.

Currently, Panaché is vulnerable to attacks by nodes that may flood the network with invalid documents. An expiration date for index entries partially addresses the issue. Malicious index servers could also generate false information. Replication of index data and checking the accuracy of queries with multiple servers could mitigate this problem at the expense of greater bandwidth consumption.

7. Conclusions

All peer-to-peer file sharing systems require efficient keyword search capabilities. Panaché provides a scalable solution to this problem by using optimizations such as Bloom filters and query ordering. Panaché provides scalability that is *O(c*m)* in each query for *m* matching documents and a small number of keywords, *c*. Using truncated queries, Panaché will use *O(c)* bandwidth, without sacrificing search quality or the ability to completely search the index. Thus, we expect that Panaché will significantly outperform Gnutella.

Panaché needs more comprehensive testing. However, based on our simulated experiments, we believe that Panaché could provide valuable search capabilities for peer-to-peer systems. We are excited about the future work to be done.

Acknowledgements

We are grateful for the guidance of Prof. Robert Morris and Thomer Gil as well as the PDOS group.

References

[1] Adar, E., Huberman B., Free Riding on Gnutella. First Monday, Peer Reviewed Journal on the Internet. Volume 5. Number 10. October 2, 2000.

[2] Brin, S., Page, L. The anatomy of a large scale hypertextual web search engine. In *7th International WWW Conference*, 1998.

[3] Broder, A., Mitzenmacher, M. Network Applications of Bloom Filters: A Survey. To appear in Allerton 2002.

[4] Cormen, T., Leiserson, C., Rivest, R., Stein, C. Introduction to Algorithms. 2nd Edition. MIT Press. 2001.

[5] Clarke, I., Sandberg, O., Wiley, B., Hong, T. Freenet: A distributed anonymous information storage and retrieval system. In *Proc. of the ICSI Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, CA, Jun. 2000.

[6] Gassend, B., Gil, T., Song, B. DINX: A Decentralized Search Engine. 6.824 project, MIT, 2001.

[7] Gnawali, O. A Keyword-Set Search System for Peer-to-Peer Networks. M.Eng thesis, MIT, 2002.

[8] Gnutella. http://gnutella.wego.com.

[9] KaZaa. http://www.kazaa.com.

[10] Markatos, E. P., Tracing a large-scale Peer to Peer System: an hour in the life of Gnutella. Technical Report 298. ICS-FORTH. In *the second IEEE International Symposium on Cluster Computing and the Grid*.

[11] Maymounkov, P., Mazières, D., Kademilia: A Peer-to-peer Information System Based on the XOR Metric. In *Proc. IPTPS'02*, Cambridge, MA, Mar. 2002.

[12] Napster. http://www.napster.com.

[13] Reynolds, P., Vahdat, A. Efficient peer-to-peer keyword searching. Technical Report 2002, Duke University, CS Department, Feb. 2002.

[14] Rowstron, A., Druschel, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM Middleware*, Heidelberg, Germany, Nov. 2001.

[15] Saroiu, S., Gummadi, P. K., Gribble, S. D., A Measurement Study of Peer-to-Peer File Sharing Systems. In *Proc. Multimedia Computing and Networking 2002 (MMCN'02)*, Jan. 2002.

[16] Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. ACM SIGCOMM'01*, San Diego, CA, Aug. 2001.

[17] Yang, B., Garcia-Molina, H. Efficient search in peer-to-peer networks. Technical Report 2001-47, Stanford University, Oct. 2001.

[18] Zhao, Y.B., Kubiatowicz, J.D., Joseph, A.D. Tapestry: An infrastructure for fault resilient wide-area location and routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, Apr. 2001.