Peer-to-Peer Architecture for Content-Based Music Retrieval on Acoustic Data

Cheng Yang*
Department of Computer Science
Stanford University
Stanford, CA 94305, U.S.A.
yangc@cs.stanford.edu

ABSTRACT

In traditional peer-to-peer search networks, operations focus on properly labeled files such as music or video, and the actual search is often limited to text tags. The explosive growth of available multimedia documents in recent years calls for more flexible search capabilities, namely search by content. Most content-based search algorithms are computationally intensive, making them inappropriate for a peer-to-peer environment. In this paper, we discuss a content-based music retrieval algorithm that can be decomposed and parallelized efficiently. We present a peer-to-peer architecture for such a system that makes use of spare resources among subscribers, with protocols that dynamically redistribute load in order to maximize throughput and minimize inconvenience to subscribers. Our framework can be extended beyond the music retrieval domain and adapted to other scenarios where resource pooling is desired, as long as the underlying algorithm satisfies certain conditions.

Categories and Subject Descriptors

H.3 [**Information Systems**]: Information Storage and Retrieval

General Terms

Algorithms, Design

Keywords

Content-based music retrieval; acoustic data; peer-to-peer; distributed; load balancing; resource pooling.

1. INTRODUCTION

Popular peer-to-peer (P2P) search networks such as *Gnutella* (`http://www.gnutella.com`) and *Morpheus* (`http://www.morpheus-os.com`) identify files based on their names and/or short text descriptions. Most research on peer-to-peer systems are also based on the assumption that short tags or "keys" are available to quickly identify files in the database [12, 14, 18]. Such an assumption ensures that each search can be carried out without consuming too much computational power at subscriber nodes (computers that participate in the network). In fact, the search processes can run in the background without being noticed by the user. On the other hand, if we want to search by content similarity, special algorithms need to be employed, which may require a large amount of CPU power. Therefore, most content-based multimedia retrieval algorithms are not designed for peer-to-peer networks.

It is desirable to have content-based multimedia search under the peer-to-peer framework, for a number of reasons. First, a peer-to-peer system gives access to a much larger database than what could be expected from a centralized server. This particular feature is a major reason why peer-to-peer is so popular today, even with its limited search capability on text tags. Secondly, fault-tolerance is easier to handle under a peer-to-peer architecture. When a node goes down, other nodes can pick up the work with little effect on the entire system. Finally, given that content-based searches are computationally intensive, it is helpful to distribute the workload over a network of available CPUs. It is important to realize that a peer-to-peer system is different from a cluster of dedicated servers: subscriber nodes on a peer-to-peer networks typically have other tasks to work on, which take absolute priority on local resources. In order to minimize inconvenience to subscribers, they should have the ability to limit the amount of resources available to peers, as well as the option to interrupt running processes at any time.

In this paper, we present a peer-to-peer architecture for the purpose of content-based music retrieval. Past research on content-based music retrieval can be divided into several categories, depending on the type of music data (symbolic vs. acoustic) and the definition of "similarity." We will give a brief overview of related work in Section 2.

Our system deals with acoustic data and uses an intuitive notion of similarity: two pieces are similar if they are fully or partially based on the same score, even if they are performed by different people or at different tempo. The input to our system is a music sound clip, while the output consists of "similar" music files retrieved. Our algorithm can be easily decomposed and parallelized, partial results can be generated throughout execution, and there are effective ways to summarize partial results into a final answer. Furthermore, the algorithm does not have to wait for results from all parallel components; it works even when certain components are terminated prematurely. All of these features make it possible to deploy our algorithm in a peer-to-peer environment. We will discuss details in Section 3.

Because of its applications in on-line music sharing, current peer-to-peer networks have become center of the controversial issue of on-line piracy, i.e., illegal sharing of copyrighted material. The system proposed in this paper is subject to a similar situation: it may potentially be used to search and share copyrighted music files. On the other hand, however, copyright owners can use the same system to enforce copyright and help pursue violators. Because the search is content-based, altering text tags or otherwise manipulating sound

*Supported by a Leonard J. Shustek Fellowship, part of the Stanford Graduate Fellowship program, and NSF Grant IIS-0085896.

WWW2003, May 20–24, 2003, Budapest, Hungary.
ACM 1-58113-680-3/03/0005.

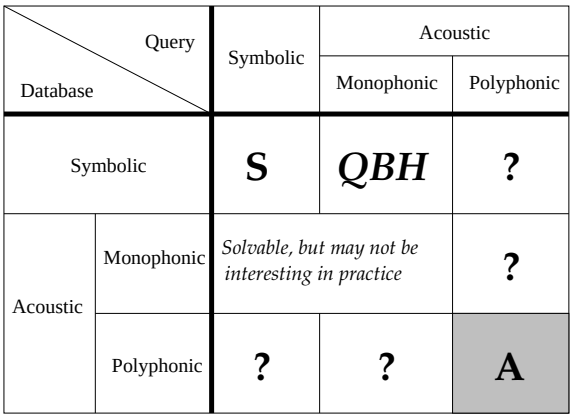

Figure 1: Classification of music retrieval systems

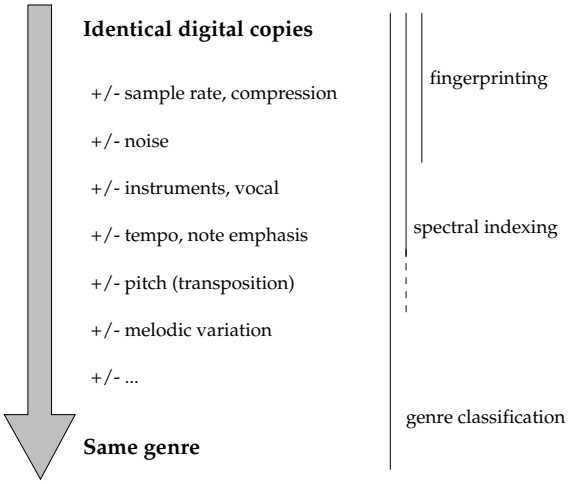

Figure 2: Different definitions of music similarity

files would not hide copyrighted material from being detected. In fact, our system can effectively discourage illegal music sharing. It can also be implemented in a way that enables peer-to-peer search and identification of copyrighted songs, but redirects download requests to authorized retailers on a commission-based model. Furthermore, the ability to do content-based search is of great interest to musicologists who would like to trace origins of certain musical phrases and to study "allusions" among different pieces.

2. CENTRALIZED MUSIC RETRIEVAL ALGORITHMS

Computer representation of music comes in two different ways. One way is symbolic representation based on musical scores. For each note in the score, it keeps track of pitch, duration (start time/ end time), strength, as well as other pertinent information. Examples of this representation include MIDI and Humdrum, with MIDI being the most popular format. Another way is based on acoustic signals, recording the audio intensity as a function of time, sampled at a certain frequency, often compressed to save space. Examples of this representation include .wav, .au, and MP3. MIDI-style data can be synthesized into audio signals easily, but there is no known algorithm to do reliable conversion in the other direction (i.e., music transcription), except in monophonic or simple polyphonic cases [2, 4, 15]. Polyphony refers to the scenario when multiple notes occur simultaneously in music. As we know, most music pieces today are polyphonic. Transcription of general polyphonic signal is extremely hard. Because of this difficulty, music retrieval algorithms that deal with acoustic data are very different from those that deal with symbolic data.

2.1 Classification of Music Retrieval Systems

Depending on music data types used in queries and the underlying database, music retrieval systems can be classified according to Figure 1, as was proposed in [20]. The categories with question marks remain open problems. Among the rest, three types are of particular interest:

- Symbolic query on a symbolic database (marked "S" in Figure 1): Both the query and the underlying database are in symbolic formats, such as MIDI and Humdrum. Retrieval problems on such symbolic data can typically be addressed by methods derived from text searching techniques. Several systems of this type have been implemented, including the Themefinder project (`http://www.themefinder.org`), where the symbolic database can be searched using pitch sequences, intervals, approximate contours, etc.

- Monophonic acoustic query on a symbolic database (marked "QBH" in Figure 1): This problem, also known as *Query by Humming* or *QBH*, has received considerable attention during the past few years [3, 6, 7, 10, 11]. In such systems, the input query is typically given as a user-hummed melody through a microphone, and the melody is analyzed and matched against a symbolic database. Human-hummed tunes are monophonic melodies and can be automatically transcribed into pitches with reasonable accuracy. In order to compensate for possible inaccuracies of human-hummed tunes, some systems use approximate contour information (up, down, etc.) or beat information to aid the retrieval process.

- Polyphonic acoustic query on a polyphonic acoustic database (marked "A" in Figure 1): This problem is our focus in this paper. Input queries are acoustic, typically short sound clips, and the goal is to find similar acoustic pieces from the database. At first sight, this area may seem even harder than those marked with question marks. However, this is not the case: solving problems in this category does not necessarily require polyphonic transcription. In fact, we do not attempt to do polyphonic transcription in our work, but try to process and match spectrograms without converting them into symbolic abstractions.

Depending on the definition of "similarity," this last category can be further divided into different directions, as shown in Figure 2. On one extreme, we can regard only identical digital copies of music as "similar," and anything else as dissimilar. With this definition, the music retrieval problem reduces to a generic data retrieval problem, which can be achieved through traditional hashing and indexing techniques. Alternatively, we can tolerate some distortions due to sample rate change, compression, or noise, and still regard the results as similar. Or, we can tolerate changes in instruments, vocal parts or tempo, and still regard the results as similar, as long as the musical "score" remains unchanged. On the other extreme, we can totally disregard the score, and define similarity as "within the same genre" - a somewhat subjective notion. The difference in similarity definitions brings about different directions in music retrieval research.

"Fingerprinting" techniques focus on finding almost-identical music recordings, while tolerating small amount of noise distortions [1]. The central idea is to extract some representative "digital signatures" from the acoustic data, which can be hashed and retrieved efficiently. Several proprietary systems have been developed at companies such as Relatable (`http://www.relatable.com`), which worked with Napster on implementing music file filters to enforce copyright protection, and *CD (`http://www.starcd.com`), whose music identification system tracks songs played on radio stations and provides users with real-time identification results.

Genre classification techniques focus on classifying music data (sometimes speech data or other forms of audio data) based on high-level properties such as energy distribution, timbre features, brightness, texture, rhythmic patterns, and so on [13, 16, 17]. Machine learning techniques such as automatic clustering are often employed during training.

Somewhere between fingerprinting approaches and genre classification approaches, there is a large area that remains mostly unexplored. The corresponding similarity definition involves similarity in musical scores, regardless of tempo, instrumentation or performance style. This definition reflects an intuitive notion of "same song" as perceived by human listeners. For the rest of this paper, we will use this similarity definition. Foote's experiments [5] demonstrated the feasibility of such tasks by matching power and spectrogram values over time using a dynamic programming method. In our previous work [19, 20], we addressed the problem with a spectral indexing technique. All of these approaches are computationally intensive, compared with text-searching systems.

2.2 The MACSIS Framework

In this section we will give an overview of a set of algorithms used in our music indexing framework. For a more elaborate treatment please refer to [20]. This set of algorithms forms a core component of our peer-to-peer network, to be discussed in Section 3.

Figure 3 shows the basic structure of our index-based retrieval system known as *Music-Audio Characteristic Sequence Indexing System*, or *MACSIS*. It consists of three phases, which are summarized below:

2.2.1 Phase 1: Characteristic Sequence Generation

Raw audio goes through a series of transformations, including Fourier Transformation, event detection and some spectral analysis, and is finally converted into a stream of *characteristic sequences*. Each characteristic sequence is a high-dimensional vector representing a short segment (typically 2 to 3 seconds) of music data. Both the audio database and the query are processed in this way. Due to space limitations, details are omitted here.

2.2.2 Phase 2: LSH Indexing and Lookup

Characteristic sequences for the audio database (generated by Phase 1) are indexed in a high-dimensional indexing scheme known as *Locality-Sensitive Hashing*, or *LSH* [8]. This scheme runs many hashing instances in parallel; in each instance, the vector is hashed by a function so that similar vectors are "likely" to be hashed to the same value, with a certain probability. At query time, characteristic sequences for queries are looked up in the LSH index to get matching items.

The design goal of such an index is to facilitate retrieval of "similar" vectors, where similarity is indicated by high correlation along a subset of dimensions. Given a query vector, the LSH is a fast probabilistic scheme that returns approximate matches with controllable false positive and false negative rates.

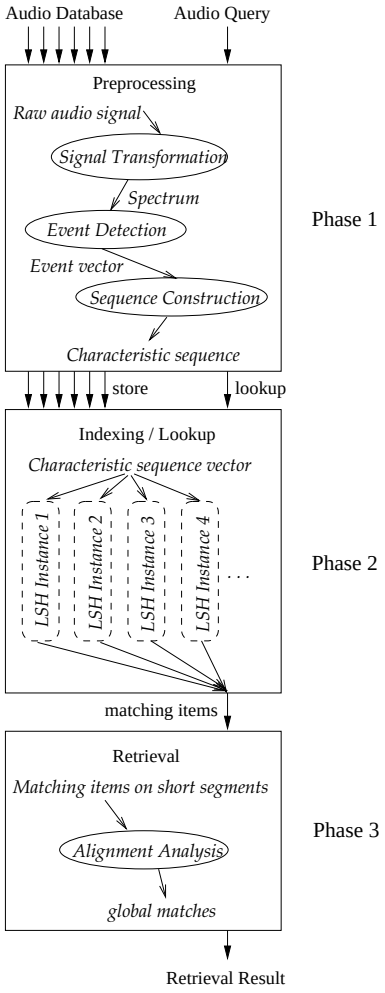

Figure 3: Structure of MACSIS

As shown in Figure 3, our LSH implementation consists of many different "hashing instances." Each hashing instance is designed to map vectors into hash values so that "similar" vectors are hashed into the same hash value with high probability. Each hashing instance has its own hash table to store hash values as well as the corresponding pointers to original vectors. All hashing instances and hash tables are independent of each other, and can be looked up in parallel. If two vectors are truly similar, their hash values are likely to agree in many of such instances. Therefore, when we process a query lookup from the hash tables, we focus on those vectors that match the query on many different hashing instances.

The key to the design is to find a family of hashing instances so that "similar" vectors can be hashed into the same hash value with high probability. In our hashing design, each raw vector first goes through a simple dimensionality-reduction routine which picks a random subset of its dimensions. Then, the sampled dimensions are normalized (to zero mean and unit variance) and passed through a low-resolution quantization grid, so that each dimension is quantized and converted into a small integer. Finally, the resulting vector of integers is passed through a universal hashing function in which each dimension is multiplied by a random weight and their sum is taken to form the final hash value, modulo the number of hash buckets.

All random parameters (dimension samples, quantization grid

lines, hashing weights) are pre-generated and fixed for each hashing instance, but vary across different instances. If two raw vectors are "similar," i.e., with a high correlation coefficient with sampled dimensions, they must be close to each other in Euclidean space with these sampled dimensions after normalization [21], so they have a good chance of being mapped to the same point after quantization. Of course they may also be unlucky and happen to lie across the quantization grid lines, even though they are close to each other. In such cases they will not be mapped to the same hash value. However, since we have many hashing instances running in parallel, there is a high probability that they would be mapped to the same hash value in several such instances. Such probabilities cannot be quantitatively analyzed without an exact mathematical definition of the similarity measure. However, the above reasoning suggests that similarity between raw vectors can be represented by the number of hashing instances that match in terms of hash values.

Different hashing instances of the same music file need not reside on the same machine. Since all hashing instances can be looked up independently of each other, they can be distributed and parallelized. Further discussion will be given in Section 3.

2.2.3 Phase 3: Alignment Analysis

During retrieval, Phase 2 finds a list of matches on characteristic sequences, representing short segments of music. We need to piece together these "partial matches" to determine which song is the best "global" match. A characteristic sequence from a query may match many different items in the database; it may even have several matches within the same music piece, since many music patterns tend to repeat themselves. Possible tempo changes add to the complexity of the problem. However, if two pieces are based on the same score, their tempo changes must be uniform in time. In other words, if one of these two pieces is slower than the other, it must be consistently slower at every time instant.

Between the query clip and a music piece from the database, possible matches can be expressed as a set of tuples (`query-offset, matching-offset, score`) which indicates time offsets of the two matching points as well as a confidence score. The set of matching time offsets (`query-offset, matching-offset`) can be plotted on a 2-D graph. If the two pieces are indeed based on the same score, with possible uniform tempo changes, we would expect to see many of such matching points lying along a straight line, either at a 45-degree angle (if there is no tempo change) or at a different angle (if there is tempo change).

To find straight lines from these matching plots, we utilize a computer-vision technique known as Hough Transform [9]. We model the hidden straight line by the equation $r = as + b$, where r and s are the matching time offset values of two music pieces. The values of a and b are estimated in the following way: each matching tuple (s_0, r_0) votes (with its own confidence score) for a set of possible (a, b) values that satisfy $r_0 = as_0 + b$. In this case, each matching tuple votes for a straight line in (a, b) space. After all matching tuples cast their votes, the (a, b) pair that receives the highest vote becomes the winner. If the vote is large enough, then we conclude that $r = as + b$ is the hidden straight line we are trying to find. This vote can also be used to measure confidence in matching these two music pieces.

2.2.4 MACSIS Implementation

In our implementation, we take a query audio clip, convert it into characteristic sequences and feed them into the indexing engine. Intermediate matching results come back in the form of (`musicID, queryOffset, matchingOffset, score`). We then select as candidates those music pieces with a large number of matching points. For each candidate, we apply the Hough Transform to the matching points (weighted by confidence scores), and rank them based on the highest vote from the Hough Transform algorithm. The top few candidates are returned.

One way to speed up retrieval is to process only a small sample of vectors (characteristic sequences) from the query rather than the entire query. Experiments in [20] show that a small fraction is often enough to get satisfactory results.

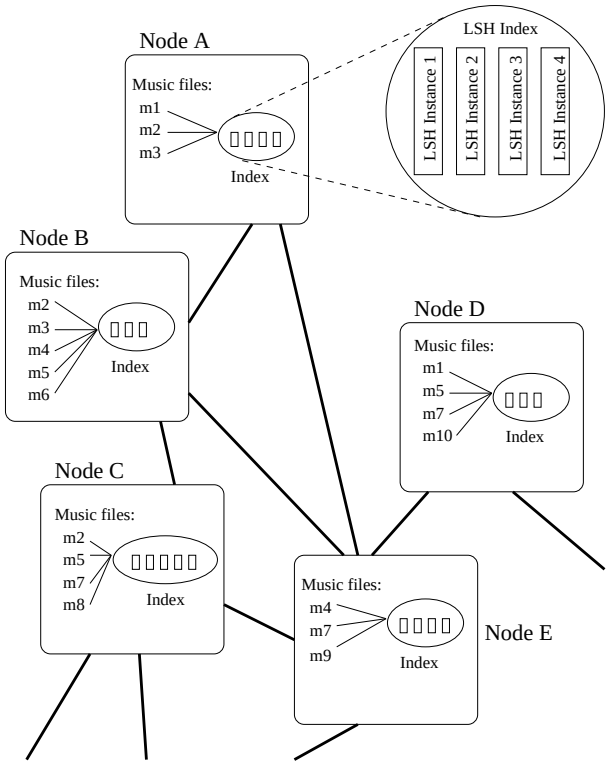

Figure 4: Illustration of peer-to-peer model for MACSIS

3. PEER-TO-PEER ARCHITECTURE

Figure 4 illustrates our peer-to-peer model for MACSIS music retrieval. Nodes are interconnected as an undirected graph (*P2P Graph*), and each node stores a collection of music files. Analysis of raw audio and conversion to characteristic sequences are done locally at each node, for both the database and the queries. While building the database, characteristic sequences for each music file are stored in multiple LSH hashing instances (each with its own set of randomized parameters). The same music file may appear in many different nodes and indexed under different sets of hashing instances. At any time, every node maintains a variable "CPU-Availability" which indicates how much CPU power it can offer to run peers' search tasks. When the node is busy with other tasks or when the owner wants to start using the node, CPU-Availability is set to zero.

A music query is issued by one node in the network (referred to as the *query node*), and processed by a set of other nodes (referred to as *response nodes*), which may include the query node itself. Because of the symmetric nature of P2P networks, any node can become a query node, and a node may be simultaneously a query node and response nodes for different queries.

A query is sent as a stream of characteristic sequences, obtained by analyzing user-input audio at the query node. Each query has

two required parameters: `expiration-time` and `search-depth`, indicating the time at which the query expires, and the maximum number of links through which the query can be passed in the P2P network graph. A query is sent from the query node to its neighbor(s), which may in turn pass it on to other neighbors of its own (while decrementing `search-depth` by 1) if `search-depth` is greater than zero.

A response node does not have to process every query vector (characteristic sequence) it receives in queries. As mentioned in the previous section, one way to speed up retrieval is to process only a fraction of such query vectors. This fraction is referred to as the query-sampling rate β. It is usually up to each response node to determine what fraction it processes, and this fraction can change over time. In the extreme case, a response node can drop the query altogether, ignoring incoming requests when it is busy.

An interesting special case is when all nodes in the P2P network store an identical set of music files, i.e., when we have a *replicated database*. This scenario may happen when a single organization has control over a cluster of nodes and wants to provide search functionality while distributing load to a set of machines. We study this case in Section 3.1.

In a general scenario, there is no requirement regarding what files each node stores, and it is totally up to the owner of each node. Furthermore, content at each node may change independently at any time. Individual nodes are not able to keep track of what files other nodes are maintaining. However, we assume that all nodes agree on a common method that hashes binary files to a short "signature" value, so that identical files must have identical signatures, and the chance of different files having identical signatures is very low. These signatures can be passed around in communication messages to indicate which files are being considered as candidate matches. We study this scenario in Section 3.2.

3.1 Replicated Database

The following are two protocols that can be used for P2P search with a replicated database. In the first one, each query is processed by one node at a time. In the second one, each query may be broken up and processed by several nodes simultaneously.

- Protocol 1: Each query is processed by one response node at a time.
  - **Query setup**: Before sending a query, the query node first polls all potential response nodes (i.e., all nodes reachable by at most `search-depth` number of hops). Nodes with available CPU will respond. Each response node decides on its query-sampling rate β, i.e., what fraction of query vectors it will process, and returns β to the query node. The choice of β may be based on its available CPU and other factors. Then, the query node picks one response node and initiates one-to-one communication in which the actual query is sent.
  - **Query processing**: The selected response node then receives query vectors, looks up in its own LSH index, and generates intermediate match tuples in the form of (`musicID, queryOffset, matchingOffset, score`). It tries to maintain its query-sampling rate β as promised, but it is under no obligation to do so; when it feels necessary to reduce its load, it will reduce β and notify the query node.
  - **Result generation**: After query processing is complete, the response node groups all intermediate matching tuples by `musicID`, selects top candidate matches based on the number of matching tuples, and performs alignment analysis to rank candidates. Candidates with the highest ranks are returned to the query node as final results.
  - **Process interruption**: During query processing, if the owner of the response node needs to use the machine, or if CPU is no longer available for some other reason, the query is interrupted immediately. The query node may also request interruption if it feels dissatisfied with the performance. Upon interruption, the response node tries to pass all intermediate match tuples back to the query node (if the response node is still alive). The query node will perform the same setup procedure as described above to select another node to continue the job. The query node then sends all intermediate match tuples to the new response node and redirects the stream of query vectors to the new node. In case of node failure at the response node, no intermediate results can be passed back, in which case the query node will select a new node to redo the work assigned to the failed node.
- Protocol 2: Each query is processed by several nodes simultaneously.
  - **Query setup**: The query node broadcasts query vectors to all potential response nodes. (It broadcasts the query to all its neighbors, which in turn relay them to their neighbors, until `search-depth` is reached.) At each potential response node, query vectors are sampled independently and buffered. Buffers are regularly checked to purge expired queries. The query node sets a goal on the overall query-sampling rate (sum of individual sampling rates at response nodes), but does not announce it to response nodes.
  - **Query processing**: Based on its CPU availability and other factors, each potential response node either ignores queries in the buffer, or picks a random one to process. In case of the latter, it looks up its LSH index and generates intermediate match tuples in the form of (`musicID, queryOffset, matchingOffset, score`), where `musicID` is based on the hash value of the music file using a pre-defined hash function agreed by all nodes. When the same music file occurs on several nodes, they all have the same ID. Intermediate match tuples are passed back to the query node. The response node also notifies the query node how many query vectors have been sampled and processed.
  - **Result generation**: The query node collects intermediate match tuples from all response nodes, groups them by `musicID`, and applies alignment analysis to top candidate matches. These steps can be pipelined, so that candidates are dynamically selected as new match tuples come in. At any time, the result of alignment analysis ranks all candidates. Candidates with the highest ranks are identified periodically and presented to the user. Actual music files are then requested from the corresponding response nodes that store them. The process stops when the overall query-sampling rate reaches the goal set by the query node, or when the expiration time arrives.
  - **Process interruption**: During query processing, if the owner of a response node needs to use the machine, or

if CPU is no longer available for some other reason, the query is interrupted immediately. If the response node is still alive, it only needs to make sure that intermediate match tuples that have already been generated are passed back to the query node. Nothing further needs to be done by the query node or by any response nodes in the network. In case of node failure at the response node, intermediate results that are not yet passed back will be wasted, but the protocol still works, with other nodes continuing their work. The query node may request interruption at all response nodes when it reaches its goal on overall query-sampling rate or when it does not wish to continue.

3.2 General P2P Network

In a general P2P network, each node maintains its own collection of files, and there is no guarantee that they will store the same collection. It no longer makes sense for a query node to compute the "overall" query-sampling rates at response nodes, as each response node is dealing with a different database. Instead, our protocol expects each response node to "try its best," and have the query node passively wait for results to come in.

Because each node maintains a different database, each query may only be relevant to a small set of nodes. To improve system throughput and reduce waste, we use a two-phase protocol: in the presearch phase, a very small sample of query vectors is given to a large number of nodes, which produce preliminary search results to help estimate which nodes are likely to contain the music file we are looking for. In the actual search phase, nodes with a higher chance of a hit will put more resources into the search.

Our protocol goes as follows:

- **Presearch phase**: The query node broadcasts a small subset of query vectors to all potential response nodes. Each node responds with a preliminary search result on the small set of query vectors. The query node evaluates the matching tuples returned, and ranks response nodes based on the total confidence scores in their preliminary matches. It then notifies all response nodes of their rankings (hereafter referred to as *relevance rankings*).

- **Search phase**:

  – **Query setup**: The query node broadcasts all query vectors to all potential response nodes. At each potential response node, query vectors are sampled independently and buffered. The node's relevance ranking from the previous phase is also stored in the buffer. Buffers are regularly checked to purge expired queries.

  – **Query processing**: Based on its CPU availability and other factors, each potential response node either ignores queries in the buffer, or picks one to process. When it picks one query from the buffer, priority is based on the relevance ranking. Higher-ranked queries are picked with higher probability. During processing, the response node looks up its LSH index and generates intermediate match tuples in the form of (`musicID, queryOffset, matchingOffset, score`), where `musicID` is based on the hash value of the music file using a predefined hash function, agreed upon by all nodes. When the same music file occurs on several nodes, they all have the same ID. Intermediate match tuples are passed back to the query node.

  – **Result generation**: The query node collects intermediate match tuples from all response nodes, groups them by `musicID`, and applies alignment analysis to top candidate matches. These steps can be pipelined, so that candidates are dynamically selected as new match tuples come in. At any time, the result of alignment analysis ranks all candidates. Candidates with the highest ranks are identified periodically and presented to the user. Actual music files are then requested from the corresponding response nodes that store them.

  – **Process interruption**: During query processing, if the owner of a response node needs to use the machine, or if CPU is no longer available for some other reason, the query is interrupted immediately. If the response node is still alive, it only needs to make sure that intermediate match tuples that have already been generated are passed back to the query node. Nothing further needs to be done by the query node or by any response nodes in the network. In case of node failure at the response node, intermediate results that are not yet passed back will be wasted, but the protocol still works, with other nodes continuing their work. The query node may request interruption at all response nodes when the user is satisfied with the partial result and stops the query.

3.3 Relevance Feedback

Regardless of the protocol used, real-time relevance feedback from users can always be added to guide the retrieval process. As partial results are returned and presented to the user, the user can point out incorrect matches, which can be communicated to all response nodes for removal from consideration (assuming the current query has not yet finished), or included with subsequent "refinement" queries. The user can also point out correct matches that may be added to existing query vectors, for the current query or for subsequent queries.

3.4 Other Applications

The P2P protocols presented in this section are not limited to the music search domain. In fact, they can be applied to many other problems where parallelization and resource pooling are desired, as long as the underlying algorithm satisfies the following conditions:

1. The algorithm can be decomposed safely and efficiently into multiple components running on different machines.
2. When any individual component is interrupted, it can be terminated or migrated without affecting other components.
3. The algorithm has a mechanism to produce incremental results based on a subset of individual components' responses.
4. The algorithm can work even if some of the parallel components fail or never respond.

The MACSIS music retrieval framework satisfies all these conditions.

4. ANALYSIS AND EXPERIMENTS

Table 1 compares certain features among the three protocols proposed in the previous section.

In Protocol 1 for replicated databases, only one response node is involved for each query, but different queries may be processed by different nodes. Final results are assembled at the response node. There is no redundancy here. However, when the response node is

	Replicated Database		General P2P
	Protocol 1	Protocol 2	2-Phase Protocol
# of concurrent response nodes	one	multiple	multiple
Granularity of parallelization	per query	per lookup	per lookup
Result assembly location	response node	query node	query node
Possible redundant work	no	yes	yes
Handling for process interruption	Migrate process	flush result	flush result
Ability to control β	yes	yes	no
Task prioritization	FCFS	random	by relevance
Application	Load balancing		P2P search

Table 1: Comparison of different P2P protocols for MACSIS

interrupted in the middle, there is a high overhead to migrate the process to another node. For the other two protocols, each query may involve several nodes, each of which responsible for a set of index lookup operations. Intermediate results are passed back to the query node where final results are assembled. There may be redundant work with these two protocols, but process interruption is simple and efficient to handle.

Both protocols for the replicated database may be used for load balancing among a cluster of machines. Overall query-sampling rate β can be enforced. The 2-phase protocol for the general P2P model can be used for generic P2P searches. The notion of overall query-sampling rate does not apply for the general case.

When nodes have multiple pending requests, different protocols handle prioritization in different ways. Protocol 1 operates on a first-come-first-served basis; a node cannot handle other requests until it finishes the current one. The other two protocols use buffers to store pending requests. In Protocol 2, pending requests are selected at random when the CPU is free. In the 2-phased protocol for general P2P networks, each pending request has a relevance ranking determined by the pre-search phase, and priority is based on this ranking.

Simulation experiments have been conducted for a network of 100 nodes. Nodes are represented by threads which generate simulated "events" according to the protocols. Each node is connected to a random set of 10-50 other nodes, and each link has a random delay of up to 0.1 second to simulate actual network delays. A total of 1000 music files are distributed randomly among all nodes, where each file appears p times. p controls the degree of overlap and is in the range of $(0, 100]$. When $p = 100$, we have a fully replicated database.

Changes in CPU availability are simulated by slowing down event generation at individual nodes. We defining a constant α ($0 \leq \alpha \leq 1$) to control average CPU availability. When $\alpha = 0$, all nodes are unavailable and nothing gets done. When $\alpha = 1$, all nodes are available for search tasks.

In our previous work [20], we tested the MACSIS system using a centralized search engine. Retrieval accuracy is defined as the probability of the "correct" answer appearing among the top five matches. This accuracy ranges between 85% and 90% when query-sampling rate β is 0.1. The average retrieval accuracy A changes as a function of β, and can be modeled approximately by $A = 1 - e^{-20\beta}$. On the other hand, execution time T (in seconds) per query is a linear function of β and can be modeled approximately by $T = 5\beta$. These approximation models are used by our simulation experiments here, where each thread simulates the performance of MACSIS algorithm on each node without actually running it. In our simulations, CPU availability at each node affects its execution time linearly.

Figure 5 shows the retrieval rate and running time for a repli-

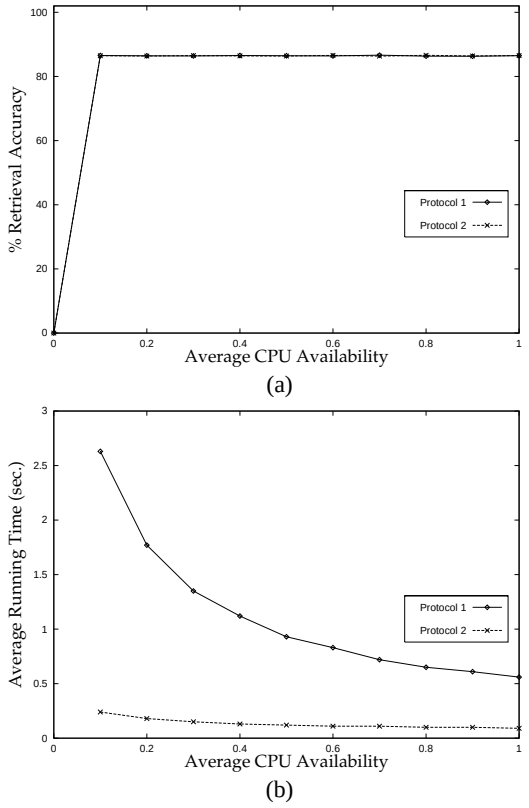

Figure 5: Experiments on replicated database search with unlimited running time

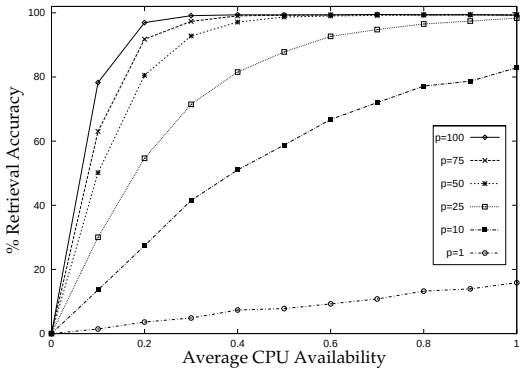

Figure 6: Experiments on general P2P search with limited running time

cated database ($p = 100$), assuming a fixed total β value (overall query-sampling rate) of 0.1 and query expiration time at infinity. Two lines represent two different protocols used. For each protocol, a total number of 200 queries are simulated. Horizontal axis is the average CPU availability. Both protocols give almost-identical curves for retrieval accuracy (with small fluctuations due to randomized behavior). When average CPU availability is above zero, performance is as good as a centralized system. Protocol 2 runs much faster than Protocol 1, because Protocol 2 leads to highly parallelized lookup.

In Figure 6, we consider the general case where each node has its own collection of music files. We vary the degree of overlap p in the range of $(0, 100]$ and use our 2-phase protocol to perform P2P search. Each query has a fixed expiration time of 0.4 second. Figure 6 shows the average retrieval accuracy as a function of CPU availability and degree of overlap. For each data point, a total of 200 queries are simulated and average results are taken. Because of limited running time, not all queries can be completed in full. Also, not every node contains a match to the query. The experimental results shown here are consistent with intuition: higher CPU availability leads to a higher retrieval rate; a higher degree of overlap gives better performance, because the more the overlap is, the easier it is to find available nodes to carry out the search.

5. SUMMARY AND FUTURE WORK

We have discussed a content-based music retrieval system that searches and retrieves audio files based on the human-perceived notion of "same song." The system can be decomposed and parallelized, and we have demonstrated how to build the system into a peer-to-peer search network. Although the search itself may be computationally intensive, our P2P design ensures that the search does not inconvenience users; only spare resources at individual nodes are utilized, and special steps are taken to interrupt processes when users need to reclaim the nodes to run other tasks.

We have presented three different protocols to do content-based music retrieval in P2P environments. Two of them are for replicated databases and can be used for load balancing among a cluster of servers. The third one is for a generic P2P architecture, which provides great flexibility and efficient execution under various circumstances. These protocols are not limited to the music search domain. In fact, they can be applied to many other domains, as long as the underlying algorithm satisfies certain conditions.

One limitation of the protocols is that they assume all nodes in the P2P network are cooperative. It is expected that all nodes will follow the specified protocols and not deviate from them. There is no mechanism to protect the system from malicious nodes who may generate an excessive amount of load or who may deliberately respond to queries with incorrect results. One future direction is to modify the protocols to identify malicious activities and to maintain stability of the network.

One way to improve performance is to categorize all music files into different genres (either manually or with the help of automated techniques). When a query is determined to be of a particular genre, only those music files within the same genre need to be searched.

Another possible future direction is to build an economic model in which all nodes have the incentive to provide search functions to peers, and can get proper reward for offering service to others. This model will eventually lead to a commercial deployment of the system.

6. REFERENCES

[1] E. Allamanche, J. Herre, O. Hellmuth, B. Fröba, T. Kastner and M. Cremer, "Content-based Identification of Audio Material Using MPEG-7 Low Level Description," in *International Symposium on Music Information Retrieval*, 2001.

[2] J. P. Bello, G. Monti and M. Sandler, "Techniques for Automatic Music Transcription," in *International Symposium on Music Information Retrieval*, 2000.

[3] S. Blackburn and D. DeRoure, "A Tool for Content Based Navigation of Music," in *Proc. ACM Multimedia*, 1998.

[4] J. C. Brown and B. Zhang, "Musical Frequency Tracking using the Methods of Conventional and 'Narrowed' Autocorrelation," *J. Acoust. Soc. Am.* 89, pp. 2346-2354. 1991.

[5] J. Foote, "ARTHUR: Retrieving Orchestral Music by Long-Term Structure," in *International Symposium on Music Information Retrieval*, 2000.

[6] A. Ghias, J. Logan, D. Chamberlin and B. Smith, "Query By Humming – Musical Information Retrieval in an Audio Database," in *Proc. ACM Multimedia*, 1995.

[7] G. Haus and E. Pollastri, "An Audio Front End for Query-by-Humming Systems," in *International Symposium on Music Information Retrieval*, 2001.

[8] P. Indyk and R. Motwani, "Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality," in *Proc. 30th Symposium on Theory of Computing*, 1998.

[9] R. Jain, R. Kasturi and B. G. Schunck, *Machine Vision*, McGraw-Hill, 1995.

[10] N. Kosugi, Y. Nishihara, T. Sakata, M. Yamamuro and K. Kushima, "A Practical Query-By-Humming System for a Large Music Database," in *ACM Multimedia*, 2000.

[11] R. J. McNab, L. A. Smith, I. H. Witten, C. L. Henderson and S. J. Cunningham, "Towards the digital music library: Tune retrieval from acoustic input," in *Proc. ACM Digital Libraries*, 1996.

[12] S. Ratnasamy, P. Francis, M. Handley, R. Karp and S. Shenker, "A scalable content-addressable network," in *Proc. ACM SIGCOMM*, 2001.

[13] E. D. Scheirer, *Music-Listening Systems*, Ph. D. dissertation, Massachusetts Institute of Technology, 2000.

[14] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *ACM SIGCOMM*, 2001.

[15] A. S. Tanguiane, *Artificial Perception and Music Recognition*, Springer-Verlag, 1993.

[16] G. Tzanetakis, G. Essl and P. Cook, "Automatic Musical Genre Classification of Audio Signals," in *International Symposium on Music Information Retrieval*, 2001.

[17] E. Wold, T. Blum, D. Keislar and J. Wheaton, "Content-Based Classification, Search and retrieval of audio," in *IEEE Multimedia*, 3(3), 1996.

[18] B. Yang and H. Garcia-Molina, "Improving search in peer-to-peer networks," in *Proc. International Conference on Distributed Computing Systems*, 2002.

[19] C. Yang, "MACS: Music Audio Characteristic Sequence Indexing for Similarity Retrieval," in *IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*, 2001.

[20] C. Yang, "Efficient Acoustic Index for Music Retrieval with Various Degrees of Similarity," in *ACM Multimedia*, 2002.

[21] C. Yang and T. Lozano-Pérez, "Image Database Retrieval with Multiple-Instance Learning Techniques," *Proc. International Conference on Data Engineering*, 2000.