Comparing Hybrid Peer-to-Peer Systems

Beverly Yang Hector Garcia-Molina
{byang, hector}@cs.stanford.edu
Computer Science Department, Stanford University

Abstract

"Peer-to-peer" systems like Napster and Gnutella have recently become popular for sharing information. In this paper, we study the relevant issues and tradeoffs in designing a scalable P2P system. We focus on a subset of P2P systems, known as "hybrid" P2P, where some functionality is still centralized. (In Napster, for example, indexing is centralized, and file exchange is distributed.) We model a file-sharing application, developing a probabilistic model to describe query behavior and expected query result sizes. We also develop an analytic model to describe system performance. Using experimental data collected from a running, publicly available hybrid P2P system, we validate both models. We then present several hybrid P2P system architectures and evaluate them using our model. We discuss the tradeoffs between the architectures and highlight the effects of key parameter values on system performance.

1 Introduction

In a *peer-to-peer* system (P2P), distributed computing nodes of equal roles or capabilities exchange information and services directly with each other. Various new systems in different application domains have been labeled as P2P: In Napster [6], Gnutella [2] and Freenet [1], users directly exchange music files. In instant messaging systems like ICQ [3], users exchange personal messages. In systems like Seti-at-home [9], computers exchange available computing cycles. In preservation systems like LOCKSS [5], sites exchange storage resources to archive document collections. Every week seems to bring new P2P startups and new application areas.

All these companies and startups tout the big advantage of P2P: the resources of many users and computers can be brought together to yield large pools of information and significant computing power. Furthermore, because computers communicate directly with their peers, network bandwidth is better utilized. However, there are often inherent drawbacks to P2P solutions precisely because of their decentralized nature. For example, in Gnutella, users search for files by flooding the network with queries, and having each computer look for matches in its local disk. Clearly, this type of solution may have difficulty scaling to large numbers of sites or complex queries. In Napster, on the other hand, users cannot search for files globally; they are restricted to searching on a single server that has only indexed a fraction of the available files.

Our goal is to study the scalability and functionality of P2P architectures, in order to understand the tradeoffs. Since we cannot possibly study *all* P2P systems at once, in this our initial paper we focus on *data-sharing, hybrid P2P systems.* The goal of a data-sharing system is to support search and exchange files (e.g., MP3s) found on user disks. In a data-sharing *pure* P2P system, all nodes are equal and no functionality is centralized. Examples of file-sharing pure P2P systems are Gnutella and Freenet, where every node is a "servent" (both a client and a server), and can equally communicate with any other connected node.

However, the most widely used file-sharing systems, such as Napster and Pointera([8]), do not fit this definition because some nodes have special functionality. For example, in Napster, a server node indexes files held by a set of users. (There can be multiple server nodes.) Users search for files at a server, and when they locate a file of interest, they download it directly from the peer computer that holds the file. We call these types of systems *hybrid* because elements of both pure P2P and client/server systems coexist. Currently, hybrid file-sharing systems have better performance than pure systems because some tasks (like searching) can be done much more efficiently in a centralized manner.

Even though file-sharing hybrid P2P systems are hugely popular, there has been little scientific research done on them (see Section 2), and many questions remain unanswered. For instance, what is the best way to organize indexing servers? Should indexes be replicated at multiple servers? What types of queries do users typically submit in such systems? How should the system deal with users that are disconnected often (dial-in phone lines)? How will systems scale in the future when user interests and hardware capacity evolve? How do different query patterns affect performance of systems from different application domains?

In the paper we attempt to answer some of these questions. In particular, the main contributions we make in this paper are:

- We present (Section 3) several architectures for hybrid P2P servers, some of which are in use in existing P2P systems, and others which are new (though based on well-known distributed computing techniques).
- We present a probabilistic model for user queries and result sizes. We validate the model with data collected from an actual hybrid P2P system run over a span of 8 weeks. (Sections 4 and 5.)
- We develop a model for evaluating the performance of P2P architectures. This model is validated via experiments using an open-source version of Napster [7]. Based on our experiments, we also derive base settings for important parameters (e.g., how many resources are consumed when a new user logs onto a server). (Sections 5 and 6.)
- We provide (Section 7.1) a quantitative comparison of file-sharing hybrid P2P architectures, based on our query and performance models. Because both models are validated on a real music-sharing system, we begin experiments by focusing on systems in the music domain.
- We project (Section 7.2) future trends in user and system characteristics, analyzing how music-sharing systems will perform in response to future demands.
- We provide (Section 7.3) a comparison of strategies in domains other than music-sharing, showing how our models can be extended to a wide range of systems.

We note that P2P systems are complex, so the main challenge is in finding query and performance models that are simple enough to be tractable, yet faithful enough to capture the essential tradeoffs. While our models (described in Sections 4 and 6) contain many approximations, we believe they provide a good and reliable understanding of both the characteristics of P2P systems, and the tradeoffs between the architectures. Sections 5 and 6 discuss in further detail the steps we took to validate our models.

We also note that the only current, available experimental data on hybrid P2P systems are in the music-sharing domain. Hence, because it is important to have a validated model as a starting point for analysis, our experimental results in Section 7 begin with scenarios from Napster and other music-sharing systems. However, the models presented in this paper are designed to be general, and we show in Section 7.3 how they are applied to domains beyond music-sharing as well.

Finally, note that by studying hybrid P2P systems, we do not take any position regarding their *legality*. Some P2P systems like Napster are currently under legal inquiry by music providers (i.e., RIAA), because they allegedly encourage copyright violations. Despite their questioned legality, current P2P systems have demonstrated their value to the community, and we believe that the legal issues will be resolved, e.g., through the adoption of royalty charging mechanisms.[1] Thus, P2P systems will continue to be used, and should be carefully studied.

2 Related Work

Several papers discuss the design of a P2P system for a specific application without a detailed performance analysis. For example, reference [1] describes the Freenet project which was designed to provide anonymity to users and to adapt to user behavior. Reference [12] describes a system that uses the P2P model to address the problems of survivability and availability of digital information. Reference [15] describes a replicated file-system based on P2P file exchanges.

Adar and Huberman conducted a study in [10] on user query behavior in Gnutella, a "pure" P2P system. However, their focus was on anonymity and the implications of "freeloading" user behavior on the robustness of the Gnutella community as a whole. There was no quantitative discussion on performance, and the study did not cover hybrid P2P systems.

Performance issues in hybrid file-sharing P2P systems have been compared to issues studied in information retrieval (IR) systems, since both systems provide a lookup service, and both use inverted lists. Much work has been done on optimizing inverted list and overall IR system performance (e.g., [18, 26]). However, while the IR domain has many ideas applicable to P2P systems, there are differences between the two types of systems such that many optimization techniques cannot be directly applied. For example, IR and P2P systems have large differences in update frequency. Large IR systems with static collections may choose to rebuild their index at infrequent intervals, but P2P systems experience updates every second, and must keep the data fresh. Common practices such as compression of indexes (e.g., [19]) makes less sense in the dynamic P2P environment. While some work has been done in incremental updates for IR systems (e.g., [11, 25]),

[1]The recent deal between the music label BMG and Napster seems to point in this direction.

the techniques do not scale to the rate of change experienced by P2P systems.

Research in cooperative web caching (e.g., [13, 20]) has also led to architectures similar to the ones studied in this paper. However, their models of performance are very different due to several key differences between their functionality and context. First, most web caches search by key (i.e., URL), making hash-based cooperative web caches (e.g., [17]) more effective than hash-based P2P systems, which must allow multi-term keyword queries. Second, URL "queries" in web caches map to exactly one page, whereas queries in file-sharing systems return multiple results, and the number of results returned is an important metric of system effectiveness. In addition, modeling the probability of getting n hits in s servers is very different from modeling the probability of a single hit in s caches. Third, there is no concept of logins or downloads in web caching, which is a large part of P2P system performance. Fourth, bandwidth consumer by transferring a page from cache to client is an important consideration in cooperative web caching, but not in P2P systems. Finally, user behavior in the Web context is fairly well understood, whereas we will see how user behavior in P2P systems may vary widely depending on system purpose.

Several of the server architectures we present in the next section either exist in actual P2P systems, or draw inspiration from other domains. In particular, the "chained" architecture is based on the OpenNap [7] server implementation, and resembles the "local inverted list" strategy [22] used in IR systems, where documents are partitioned across modes and indexed in subsets. The "full replication" architecture uses the NNTP [16] approach of fully replicating information across hosts, and a variation of the architecture is implemented by the Konspire P2P system [4]. The "hash" architecture resembles the "global inverted list" strategy [22] used in IR systems, where entire inverted lists are partitioned lexicographically across nodes. Finally, the "unchained" architecture is derived from the current architecture in use by Napster [6].

3 Server Architecture

We begin by describing the basic concepts in a file-sharing, hybrid P2P system, based on the OpenNap [7] implementation of a file-sharing service.[2] We then describe each of the architectures in general terms. . Figure 1 shows the components in the basic hybrid P2P system and how they interact.

General Concepts. There are three basic actions supported by the system: login, query and download.

On *login*, a client process running on a user's computer connects to a particular server, and uploads metadata describing the user's library. A *library* is the collection of files that a user is willing to share. The *metadata* might include file names, creation dates, and copyright information. The server maintains an index on the metadata of its client's files. For now, we assume the index takes the form of inverted lists [23]. Every file's metadata is considered a document, with the text

[2]We are not following the exact protocol used by OpenNap, but rather use a simplified set of actions that represent the bulk of the activity in the system.

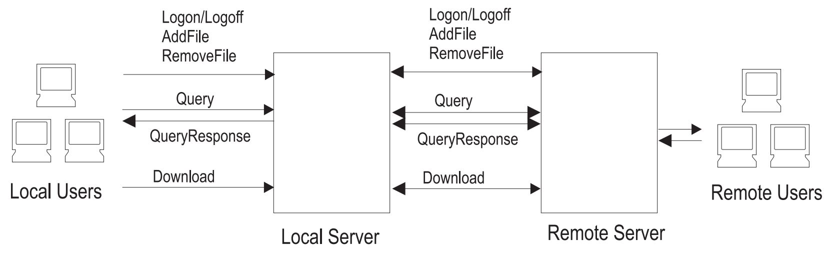

Figure 1: System Components and Messages

of the file name, author name, and so on, being its content. The server also maintains a table of user *connection information*, describing active connections (e.g., client IP address, line speed).[3] By logging on, the user is now able to query its server, and is allowing other users to download her files.

A system may contain multiple servers, but a user is logged in and connected to only one server, its *local server*. To that user, all other servers are considered *remote servers*. From the perspective of one server, users logged on to it directly are its *local users*. Depending on the architecture, servers may index the library information of both local and remote users.

A *query* consists of a list of desired words. When a server receives a query, it searches for matches in its index. The server sets a maximum number of results to return for any query, and a query is said to be *satisfied* if the maximum number of results are returned. A query is processed by retrieving the inverted lists for all its words, and intersecting the lists to obtain the identifiers of the matching documents (user files). Clearly, other query and indexing schemes are possible (e.g., relational queries), and we discuss how to evaluate these schemes with our models in Section 7.3.

The user examines query results, and when she finds a file of interest, her client directly contacts the client holding the file, and *downloads* the file. After a successful download, or after a file is added through some other means, the client notifies the local server of the new addition to the library. The local server will add this information to its index. The local server is also notified when local files are deleted. Depending on architecture, remote servers may also be notified of the addition/deletion of local files.

Upon logoff, the local server updates the index to indicate that the user's files are no longer available. Again, remote servers may have to update their indices as well, depending on architecture. The options for handling logoffs are discussed in the next subsection.

Most hybrid file-sharing P2P systems offer other types of services to users other than just file sharing, such as chat rooms, hot lists, etc. These services are important in building community and keeping users attached to the main service. However, for our study we do not consider the effects of these activities on the system, as our experiments show that they do not significantly contribute to the workload.

As we discuss and study our hybrid P2P architectures, we will introduce a number of descriptive parameters. We show all parameters and their base values in Table 1, Table 2, and Table 3, even

[3]Often, clients use dial-in connections, so their IP address can vary from connection to connection.

Parameter Name	Default Value	Description
FilesPerUser	168	Average files per user library
FracChange	.1	Average percent of a user's library that is changed offline
WordsPerFile	10	Average words per file title
WordsPerQuery	2.4	Average keywords per query
CharPerWord	5	Average characters per word
QueryPerUserSec	.000833	Average number of queries per second per user
QueryLoginRatio	.45	Ratio of queries to logins per second per user
QueryDownloadRatio	.5	Ratio of queries to downloads per second per user
ActiveFrac	.05	Percent of the total user population that is active at any given time
λ_f	100	Inverse frequency skew (to be described in Section 4)
r	4	Query skew to occurrence skew ratio (to be described in Section 4)

Table 1: User-Defined Parameters

Parameter Name	Default Value	Description
LANBandwidth	80 Mb/s	Bandwidth of LAN connection in Mb/s
WANBandwidth	8 Mb/s	Bandwidth of WAN connection in Mb/s
CPUSpeed	800 MHz	Speed of processor in MHz
NumServers	5	Number of servers in the system.
MaxResults	100	Maximum number of results returned for a query
User-Server Network	WAN	The type of network connection between users and servers
Server-Server Network	LAN	The type of network connection between servers

Table 2: System-Defined Parameters

though many of the parameters and their values will be described in later sections. Parameters are divided into user-dependent parameters (Table 1) – those parameters that describe characteristics of user behavior, system parameters (Table 2) – those parameters that determine available system resources, and derived parameters (Table 3) – those parameters that are derived from other user and system parameters. The last derived parameter, *UsersPerServer*, is the value we want to maximize for each server. Our performance model calculates the maximum users per server supportable by the system, given all other parameters.

Batch and Incremental Logins. In current hybrid P2P systems, when a user logs on, metadata on her entire library is uploaded to the server and added to the index. Similarly, when she logs off, all of her library information is removed from the index. At any given time, only the libraries of connected, or active, users are in the index. We call this approach the *batch* policy for logging in. While this policy allows the index to remain as small and thereby increases query efficiency, it also generates expensive update activity during login and logoff.

An alternative is an *incremental* policy where user files are kept in the index at all times. When

Parameter Name	Description
ExServ	Expected number of servers needed to satisfy a query
ExTotalResults	Expected number of results returned by all servers
ExLocalResults	Expected number of results returned by the local server
ExRemoteResults	Expected number of results returned by remote servers
UsersPerServer	Number of users logged on to each server

Table 3: Derived Parameters

	Chained	**Full Replication**	**Hash**	**Unchained**
Login	Index library metadata.	Index library metadata, forward metadata to all remote servers.	Hash metadata words, forward metadata to appropriate servers.	Index library metadata.
Query	Search local index. While not satisfied, forward query to remote servers.	Search local index.	Hash query terms, retrieve necessary inv. lists, and merge.	Search local index.
Download	Update local index.	Update local index, forward metadata to all remote servers.	Hash metadata words, forward metadata to appropriate servers.	Update local index.

Table 4: Architecture Summary

a user logs on, only files that were added or removed since the last logoff are reported. If few user files change when a user is offline, then incremental logins save substantial effort during login and logoff. (The parameter *FracChange* tells us what fraction of files change when a user if offline.) However, keeping file metadata of *all* users requires filtering query results so that files belonging to inactive users are not returned. This requirement creates a performance penalty on queries. Also notice that a user must always reconnect to the same server, at least with the chained servers we have described so far. This restriction may be a disadvantage in some applications.

Chained Architecture. In this architecture, the servers form a linear chain that is used in answering queries. When a user first logs on, only the local server adds library metadata to its index; remote servers are unaffected. When a user submits a query, the local server attempts to satisfy the query alone. However, if the local server cannot find the maximum number of results, it will forward the query to a remote server along the chain. The remote server will return any results it finds back to the first server, which will then forward the results to the user. The local server continues to send the query out to the remaining remote servers in the chain until the maximum number of results have been found, or until all servers have received and serviced the query. In this architecture, logins and downloads are fast and scalable because they affect only the local server of a user. However, queries are potentially expensive if many servers in the chain are needed to satisfy the query.

Full Replication Architecture. Forwarding queries to other servers can be expensive: each new server must process the query, results must be sent to the originating server, and the results must be merged. The full replication architecture avoids these costs by maintaining on each server a complete index of all user files, so all queries can be answered at a single server. Even with incremental logins, users can now connect to any server. The drawback, however, is that all logins must now be sent to every server, so that every server can maintain their copy of the index (and of the user connection information). Depending on the frequency ratio of logins to queries, and on the size of the index, this may or may not be a good tradeoff. If servers are connected by a broadcast network, then login propagation can be more efficient.

Hash Architecture. In this scheme, metadata words are hashed to different servers, so that a given server holds the complete inverted list for a subset of all words. We assume words are hashed in such a way that the workload at each server is roughly equal. When a user submits a query, we assume it is directed to a server that contains the list of at least one of the keywords. That server then asks the remaining servers involved for the rest of the inverted lists. When lists arrive, they are merged in the usual fashion to produce results. When a client logs on, metadata on its files (and connection information) must be sent to the servers containing lists for the words in the metadata. Each server then extracts and indexes the relevant words.

This scheme has some of the same benefits of full replication, because a limited number of servers are involved in each query, and remote results need not be sent between servers. Furthermore, only a limited number of servers must add file metadata on each login, so it is less expensive than full replication for logins. The main drawback of the hash scheme is the bandwidth necessary to send lists between servers. There are several ways to make this list exchange more efficient [24]; we describe one such technique in Section 6.

Unchained Architecture. The "unchained" architecture simply consists of a set of independent servers that do not communicate with each other. A user who logs on to one server can only see the files of other users at the same local server. This architecture, currently used by Napster, has a clear disadvantage of not allowing users to see all other users in the system. However, it also has a clear advantage of scaling linearly with the number of servers in the system. Though we cannot fairly compare this architecture with the rest (it provides partial search functionality), we still study its characteristics as a "best case scenario" for performance.

4 Query Model

To compare P2P architectures, we need a way to estimate the number of query results, and the expected number of servers that will have to process a query. In this section we describe a simple query model that can be used to estimate the desired values.

We assume a universe of queries $q_1, q_2, q_3, \ldots$. We define two probability density functions over this universe:

- g – the probability density function that describes query popularity. That is, $g(i)$ is the probability that a submitted query happens to be query q_i.
- f – the probability density function that describes query "selection power". In particular, if we take a given file in a user's library, with probability $f(i)$ it will match query q_i.

For example, if $f(1) = 0.5$, and a library has 100 files, then we expect 50 files to match query q_1. Note that distribution g tells us what queries users like to submit, while f tells us what files users like to store (ones that are likely to match which queries).

Calculating ExServ for the Chained Architecture. Using these two definitions we now compute *ExServ*, the expected number of servers needed in a chained architecture (Section 3) to obtain the desired $R = \mathit{MaxResults}$ results. Let $P(s)$ represent the probability that exactly s

servers, out of an infinite pool, are needed to return R or more results.

The expected number of servers is then:

$$ExServ = \sum_{s=1}^{k} s \cdot P(s) + \sum_{s=k+1}^{\infty} k \cdot P(s) \tag{1}$$

where k is the number of servers in the actual system. (The second summation represents the case where more than k servers from the infinite pool were needed to obtain R results. In that case, the maximum number of servers, k, will actually be used.)

In Appendix A.2 we show that this expression can be rewritten as:

$$ExServ = k - \sum_{s=1}^{k-1} Q(s \cdot \mathit{UsersPerServer} \cdot \mathit{FilesPerUser}) \tag{2}$$

Here $Q(n)$ is the probability that R or more query matches can be found in a collection of n or fewer files. Note that $n = s\cdot$ *UsersPerServer* $\cdot$ *FilesPerUser* is the number of files found on s servers.

To compute $Q(n)$, we first compute $T(n,m)$, the probability of exactly m answers in a collection of exactly n files. For a given query q_i, the probability of m results can be restated as the probability of m successes in n Bernoulli trials, where the probability of a success is $f(i)$. Thus, $T(n,m)$ can be computed as:

$$T(n,m) = \sum_{i=0}^{\infty} g(i) \Big(\binom{n}{m} (f(i))^m (1 - f(i))^{n-m} \Big). \tag{3}$$

$Q(n)$ can now be computed as the probability that we do not get $0, 1, ...$ or $R-1$ results in exactly n files, or

$$Q(n) = 1 - \sum_{m=0}^{R-1} T(n,m). \tag{4}$$

Figure 2 illustrates $Q(n)$ for several cases where $r \geq 1$. For this graph, $R = \mathit{MaxResults} = 100$, and $\lambda_f = 1000$. We can see that if the server(s) have few files, then it is very unlikely that the desired $R = 100$ results will be found. However, as the n increases beyond a hundred thousand, the probability $Q(n)$ increases rapidly. As r decreases, λ_g decreases and the popularity distribution gets more skewed. The very popular queries are more likely to occur, and they get more results, so the threshold R is reached sooner. Hence, curves for smaller r rise more steeply.

To illustrate our model, Figure 3 shows values for *ExServ* as a function of the number of files *per server*. For this graph, we assume $R = 100$ and $\lambda_f = 1000$. In addition, we assume there are 10 servers in the system. When the number of files per server, n, is small (e.g., less than 20,000), then a query will be serviced by all 10 servers in order to attempt to reach the threshold of R results (there is no guarantee that even in 10 servers, all R results will be found). As n grows, however, it becomes more likely to find R results in a smaller number of servers, hence *ExServ* decreases. For example, we see in the figure that if each server holds 200,000 files, and $r = 1$, then the query need

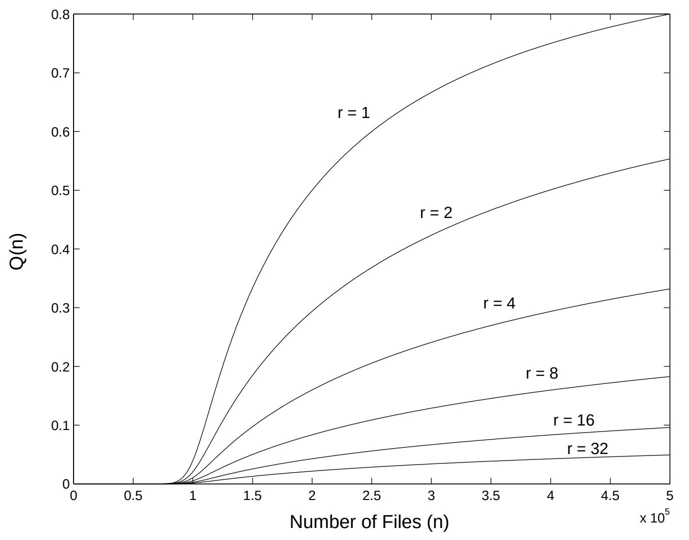

Figure 2: Q(n) distributions for $r \geq 1$

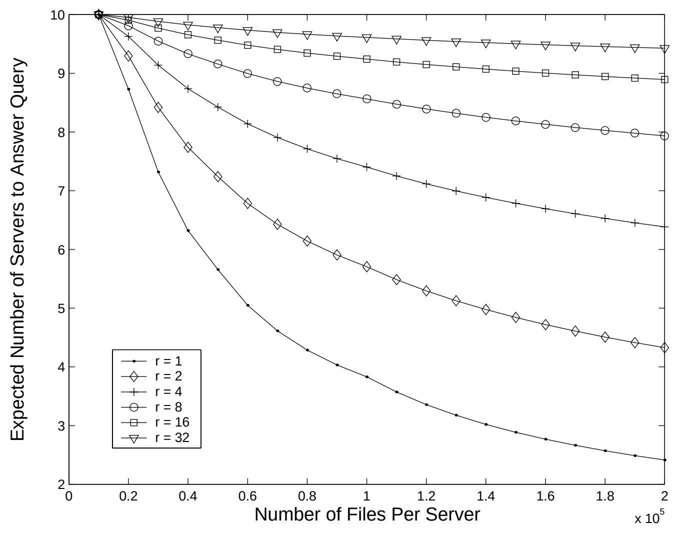

Figure 3: ExServ for $r \geq 1$

only be serviced by 2.4 servers on average before R results are found. Note that as r decreases, the threshold R is reached sooner because queries with large selection power are more likely to be queried. As a result, *ExServ* is lower.

Calculating Expected Results for Chained Architecture. Next we compute two other values required for our evaluations, *ExLocalResults* and *ExRemoteResults*, again for a chained architecture. *ExLocalResults* is the expected number of query results from a single server, while *ExRemoteResults* is the expected number of results that will be returned by a remote server. The former value is needed, for instance, to compute how much work a server will perform as it answers a query. The latter value is used, for example, to compute how much data a server receives from remote servers that assist in answering a query.

Both values can be obtained if we compute $M(n)$ to be the expected number of results returned from a collection of n files. Then, *ExLocalResults* is simply $M(y)$, where y = *UsersPerServer* $\cdot$ *FilesPerUser*. Similarly, the expected total number of results, *ExTotalResults*, is $M(k \cdot y)$. Then *ExRemoteResults* is *ExTotalResults* - *ExLocalResults* $= M(k \cdot y) - M(y)$.

In Appendix A.3 we show that

$$M(n) = \sum_{i=0}^{\infty} g(i) \Big(R - \sum_{m=0}^{R-1} \binom{n}{m} (f(i))^m (1 - f(i))^{N-m} (R - m) \Big). \tag{5}$$

Calculating Expected Values for Other Architectures. So far we have assumed a chained architecture. For full replication, *ExServ* is trivially 1, since every server contains a full replica of the index at all other servers. Because *ExServ* is 1, all results are local, and none are remote. Hence *ExRemoteResults* $= 0$, and *ExLocalResults* = *ExTotalResults* $= M(N)$, where M is defined above, and $N = s\cdot$ *UsersPerServer* $\cdot$ *FilesPerUser* is the number of files in the entire system.

For the hash architecture, again *ExRemoteResults* $= 0$ and *ExLocalResults* $= M(N)$, since all the results are found at the server that the client is connected to. *ExServ* is more difficult to calculate, however. The problem again takes the form of equation (1), but instead of using the probability $P(s)$, we want to use a different probability $P_2(s)$ that exactly s servers are "involved" in answering the query. A server is "involved" in a query if it contains at least one of the inverted lists needed to answer the query. We assume that on average, *WordsPerQuery* lists are needed to answer a query; however, it is possible for more than one list to exist at the same server. The problem of finding the probability of b lists residing at exactly i servers can be restated as the probability of b balls being assigned to exactly i bins, where each ball is equally likely to be assigned to any of the k total bins. Because the average number of words per query is typically quite small, we are able to explicitly compute the necessary values, and interpolate the values to get a close approximation in the case of fractional *WordsPerQuery*. For example, say *WordsPerQuery* $= 3$ and *NumServers*

= 5. Then the probability of all lists residing at one, or at 3 distinct servers is:

$$\text{At 1 server: } \binom{5}{1}\left(\frac{1}{5}\right)^3 = .04 \qquad \text{At 3 distinct servers: } \binom{5}{3} 3! \left(\frac{1}{5}\right)^3 = .48 \tag{6}$$

Finally, the probability of all lists residing at 2 distinct servers is 1 minus the probabilities of 1 and 3 distinct servers, which is .48. The expected number of servers involved in a query with 3 words over a set of 5 servers is therefore 2.44. Please refer to Appendix A.4 for a complete solution.

For the unchained architecture, *ExServ* is trivially 1, since queries are not sent to remote servers. *ExRemoteResults* = 0 and *ExLocalResults* = $M(n)$, where $n =$ *UsersPerServer* $\cdot$ *FilesPerUser* is the number of files at a *single* server.

Distributions for f and g. While we can use any g and f distributions in our model, we have found that exponential distributions are computationally easier to deal with and provide accurate enough results (see Section 5) in the music domain. Thus, we assume for now that $g(i) = \frac{1}{\lambda_g} e^{-\frac{i}{\lambda_g}}$. Since this function monotonically decreases, this means that q_1 is the most popular query, while q_2 is the second most popular one, and so on. The parameter λ_g is the mean. If λ_g is small, popularity drops quickly as i increases; if λ_g is large, popularity is more evenly distributed.

Similarly, we assume that $f(i) = \frac{1}{\lambda_f} e^{-\frac{i}{\lambda_f}}$. Note that this assumes that popularity and selection power are correlated. In other words, the most popular query q_1 has the largest selection power, the second most popular query q_2 has the second largest power and so on. This assumption is reasonable because in a "library-driven" system where queries are answered over users' personal collection of files, popular files are *queried* for frequently, and *stored* frequently[4]. Stored files can be obtained from the system by downloading query results, or from an external source. (For example, in the music domain, files can be obtained and stored from "ripped" CDs, and we expect a correlation between these files and what users are looking for.) However, the mean λ_f can be different from λ_g. For example, if λ_g is small and λ_f is large, popularity drops faster than selection power. That is, a rather unpopular query can still retrieve a fair number of results.

Some readers may be concerned that, under these assumptions, eventually everyone would have stored the popular files, and no one would need to query for them anymore. However, at least in the music domain, such a steady state is never reached. One reason is that users do not save all files they search for, or even all files they download. In particular, a study done by [21] shows that on average, college-age users who have downloaded over 10 songs eventually discard over 90% of the downloaded files. Hence, the high frequency of a query does not necessarily drive up the size of the result set returned. Another reason why a steady state is never reached is that user interests (and hence the ordering of queries by popularity and selectivity) vary over time. Our model only captures behavior at one instant of time.

From this point on, we will express λ_g as $r \cdot \lambda_f$, where r is the ratio λ_g/λ_f. For a given query popularity distribution g, as r decreases toward 0, selection power f becomes more evenly

[4]This characteristic also holds true for "cache-driven" systems like Freenet that use the LRU replacement strategy.

distributed, and queries tend to retrieve the same number of results regardless of their popularity.

Section 7.3 describes how the query model behaves with different distributions for f and g.

5 OpenNap Experiments

In this section we describe how we obtained statistics from a live P2P system. Using this data we validate our query model with the given assumptions and derive base values for our query parameters.

The OpenNap project is open source server software that clones Napster, one of the most popular examples of hybrid P2P systems. An OpenNap server is simply a host running the OpenNap software, allowing clients to connect to it and offering the same music lookup, chat, and browse capabilities offered by Napster. In fact, the protocol used by OpenNap follows the official Napster message protocol, so any client following the protocol, including actual Napster clients, may connect. OpenNap servers can be run independently, as in the unchained architecture, or in a "chain" of other servers, as in the chained architecture. All users connected to a server can see the files of local users on any of the other servers in the chain (a capability Napster lacks).

Our data comes from an OpenNap server run in a chain of 5 servers. Up to 400 simultaneous real users connected to our server alone, and up to 1350 simultaneous users were connected across the entire chain.

Once an hour, the server logged workload statistics such as the number of queries, logins, and downloads occurring in the past hour, words per query, and the current number of files in the server. Naturally, all statistics were completely anonymous to protect the privacy of users. The data was gathered over an 8 week period, and only data from the last six weeks was analyzed, to ensure that the data reflected steady-state behavior of the system.

The workload on our server varied throughout the experimental period depending on the time of day and day of week. However, the variation was regular and predictable enough such that we could average the numbers to get good summary information to work with.

5.1 Query Model Validation

As part of the hourly log, the server recorded how many queries in the period obtained 0 results, how many obtained 1 result, and so on. In effect, for every hour we obtained a histogram giving the frequency at which each number of results was returned by the server. For each histogram, we also recorded the number of files that were indexed at that point. The observed number of files ranged between 40,000 and 95,000 files. Clearly, the histogram depended on the number of files; when there were more files, users tended to get more results.

To increase the accuracy of our measurements, we combined 19 histograms that occurred when the server had roughly 69,000 files (actually between 64,000 and 74,000 files). Thus, the combined histogram showed us how many queries in all the relevant periods had obtained 0 results, how many had obtained 1 result, and so on. Figure 4 shows the cumulative normalized version of that

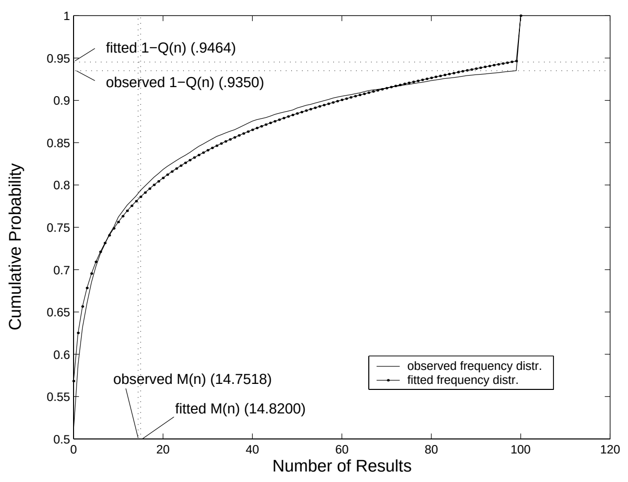

Figure 4: Observed and Fitted Frequency Distributions of Results in OpenNap

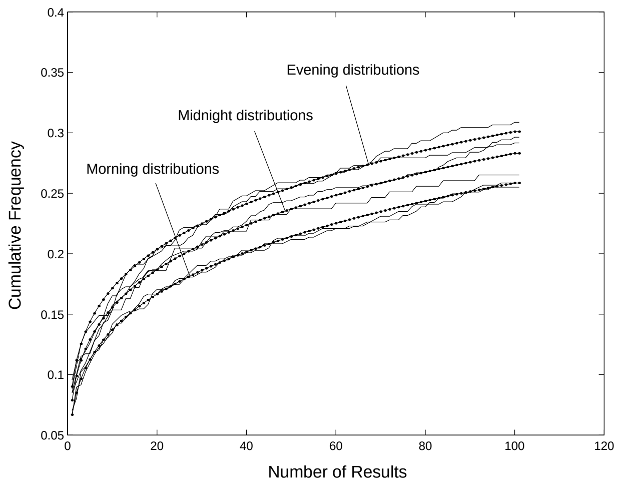

Figure 5: Observed and Fitted Frequency Distributions of Results in Napster

histogram (solid line). For example, about 87.3% of all the queries had 40 or fewer results. At 100 results we observe a jump in the distribution because at most 100 results are returned by the server. Thus, 93.5% of the queries returned 99 or fewer results, and $1-93.5\% = 6.5\%$ of the queries would have returned 100 or more results, but were forced to return only 100. Keep in mind that Figure 4 shows results for a single server. When a query returns less than 100 local results, other servers in the chain are contacted for more results, but those results are not shown in this graph.

Our next step is to fit the curve of Figure 4 to what our query model would predict. Since $T(n,m)$ is the probability that a query returns m results when run against n files (see Section 4), the cumulative distribution of Figure 4 should match the function $h(x) = \sum_{m=0}^{x} T(69000, m)$ in the range $0 \le x \le 99$. By doing a best fit between this curve and the experimental data, we obtain the parameter values $\lambda_f = 400$ and $r = 10$ (and $\lambda_g = 400 \cdot 10 = 4{,}000$). These parameters mean that the 400th query has the mean selectivity power, while the 4000th query has the mean probability of occurring. Figure 4 shows the fitted curve $h(x)$ with $\lambda_f = 400$ and $r = 10$.

With the fitted parameters, our model predicts a mean number of results of $M(69000) = 14.82$, which differs from the observed mean by only 1.22%. Similarly, our model predicts that with probability $1 - Q(69000) = 0.9464$, there will be fewer than 100 results. This only differs by 0.46% from the measured value (see Figure 4). Thus, our model clearly describes well the observed data from our OpenNap experiments.

To observe how our model worked in a different setting, we ran a second set of experiments. We took a random sample of 660 queries submitted to the OpenNap server and submitted them to Napster. We submitted the same sample of queries six times at different times of day – evening (7 pm), midnight (12 am), and morning (8 am), when the load on Napster was heaviest, lightest, and average, respectively. We recorded the number of results returned for the queries, obtained a result frequency distribution for each run, and fitted curves to the observed data.

Table 5 shows the results for these experiments, together with the λ_f and r values obtained by

Run	Number of Files	Time of Day	$M(n)$	$1-Q(n)$	λ_f	r	Fitted $M(n)$	Fitted $1-Q(n)$
1	1335411	morning	80.01	.2585	100	3.68	79.44	.2633
2	1539166	morning	80.03	.2550	100	3.68	80.21	.2538
3	1969233	evening	75.77	.3087	140	4.14	75.86	.3025
4	2038807	evening	76.33	.2963	140	4.14	76.06	.2999
5	1256387	midnight	77.38	.2917	100	3.92	77.21	.2890
6	1468911	midnight	78.17	.2830	100	3.90	77.68	.2830

Table 5: Characteristics of Napster Query Runs

curve fitting. As expected, the values for $M(n)$ and $1-Q(n)$ are closely matched by the fitted values within each run, and parameters r and λ_f are fairly close across the runs. Interestingly, the obtained λ_f and r values do not only differ from the values we obtained from OpenNap, but they also differ slightly by time of day. Note for example that, even though more files are available in the evening experiments, fewer average results are obtained by the queries (smaller M values). The tight correlation between query characteristics and time of day is highlighted in Figure 5, where the observed Napster distributions are marked in solid lines, and the fitted distributions in dotted lines.[5]

The explanation for the differences in parameters between the two systems, and between the different times of day, is that the communities that use P2P systems vary, and hence the types of files they make available (and the queries they submit) differ. Morning Napster users are not at school and wake up early, and have somewhat different interests from Napster users at midnight, who are more likely to be college students. Hence, morning and midnight query behavior differ. Similarly, since Napster is widely known, it tends to have more casual users who access popular songs. OpenNap, on the other hand, is harder to find and use, so its users tend to be more involved in the music-sharing concept and community, more technically savvy, and may not always have tastes that conform to the mainstream. As a result, query frequencies and selectivities for OpenNap are more evenly distributed, accounting for the larger λ_f and larger r. In contrast, the popularity skew of songs in Napster is probably high and heavily influenced by the latest trends. Hence, λ_f and r are relatively small.

We conclude from these observations that our query model describes the real world sufficiently well, although we need to consider (as we will do) the effect of different λ_f and r values on the performance of the various architectures.

5.2 Parameter Values

From the experimental data, we were also able to determine average values for most of our user-dependent model parameters. Table 1 summarizes these parameter values. Due to space limitations we cannot comment on each value, and how it was obtained. However, we do comment on one parameter which may be biased, and two parameters we were unable to measure directly.

[5]There is again a jump at $R = 100$ results to cumulative frequency of 1, but it is not shown because the value 1 is off the scale.

First, our measured value of *QueryLoginRatio* was 0.45, meaning roughly that on average users submit one query every other session. We were surprised by this number, as we expected users to at least submit one query per session. After analyzing our data, we discovered two main reasons for the low *QueryLoginRatio* value. First, many clients operate over unreliable modem connections. If the connection goes down, or even is slow, the client will attempt to re-login to the server. A client may thus login several times in what is a single session from the user's point of view. Second, many users are drawn to the OpenNap chat community. Users often log on to the server simply to chat or see who is online, rather than to download files. Our data shows that chat accounts for 14.6 KB/hour of network traffic – not enough to significantly impact the system performance, but enough to bias *QueryLoginRatio*. For our evaluation we initially use the measured value of 0.45, but then we consider higher values that may occur with more reliable connections or with less chat oriented systems.

Parameter *FracChange* represents the fraction of a user's library that changes between a logoff and the next login. Since in OpenNap one cannot relate a new login to a previous logoff, this value cannot be measured. A library may change offline because a user deletes files, or because she "rips" a CD. We estimate that *FracChange* = 10% of a user's 168 average number of files may be changed, mainly because files that were downloaded are played a few times and then removed to make room for new material. In Section 7.1 we perform a sensitivity analysis on parameter *FracChange* to see how critical our choice is.

Parameter *ActiveFrac* indicates what fraction of the user population is currently active. We estimate that *ActiveFrac* = 0.05 as follows. Our statistics show an average of 226 active local users at any given time, and with *QueryPerUserSec*/ *QueryLoginRatio* = .00184 logins per user per second, there are approximately 36015 logins a day. Let us assume that the average user submits 3 to 4 queries a day. This means the average user logs in *QuerysPerUserPerDay*/ *QueryLoginRatio* = 8 times a day. (The high value for logins per user per day is a direct result of the low value observed for *QueryLoginRatio*. See discussion above.) If each user logs on 8 times a day, then there are a total of approximately 4502 users in our local user base, which means on average, approximately 226/4502 = .050 of all users are active at any given time. Again, in Section 7.1 we study the impact of changing this parameter.

6 Performance Model

In this section, we describe the performance model used to evaluate the architectures. We begin by defining our system environment and the resources our model will consider. We then present the basic formulas used to calculate the cost of actions in the system. Finally, we illustrate how we put the parameters and costs together to model the performance of the chained architecture. Because of space limitations, we are only able to give detailed examples of just a portion of what our model covers. For a complete description, please refer to Appendix B.

Action	Formula for CPU instructions
Batch Login/off	$152817.6 \cdot \mathit{FilesPerUser} + 200000$
Batch Query	$(4000 + 3778) \cdot \mathit{ExTotalResults} + 100000 \cdot \mathit{ExServ} + 2000 \cdot \mathit{ExRemoteResults}$
Batch Download	222348
Incr. Login/off	$77408.8 \cdot \mathit{FilesPerUser} \cdot \mathit{FracChange} + 200000$
Incr. Query	$(4000 + 3778/\mathit{ActiveFrac}) \cdot \mathit{ExTotalResults} + 100000 \cdot \mathit{ExServ}$ $+2000 \cdot \mathit{ExRemoteResults}$
Incr. Download	222348

Table 6: Formulas for cost of actions in CPU instructions

System Environment and Resources. In our model, we assume the system consists of *NumServers* identical server machines. The machines are connected to each other via a broadcast local area network (LAN), for example, if they all belong to the same organization, or through a point-to-point wide area network (WAN). Similarly users may be connected to servers via LAN or WAN. Our system can model any of the four combinations; let us assume for the examples in the remainder of the section that servers are connected to each other via LAN, and to users via WAN.

We calculate the cost of actions in terms of three system resources: CPU cycles, inter-server communication bandwidth, and server-user communication bandwidth. If users and servers are all connected via the same network (e.g., the same LAN), then the last two resources are combined into one. For the time being, we do not take I/O costs into consideration, but assume that memory is cheap and all indexes may be kept in memory. Memory is a relatively flexible resource, and depends largely on the decisions of the system administrator; hence, we did not want to impose a limit on it. Later in Section 7, we will discuss the memory requirements of each architecture, and point out the tradeoffs one must make if a limit on memory is imposed.

Table 2 lists the default system parameter values used in our performance analysis. The bandwidth values are based on commercial enterprise level standards, while the CPU represents current high-end commodity hardware. *NumServers* and *MaxResults* were taken from the OpenNap chain settings.

CPU Consumption. Table 6 lists the basic formulas for the cost of actions in CPU instructions, for the simple architecture. In this table, we see that the cost of a query is a function of the number of total and remote results returned, the cost of logging on is a linear function of the size of the library, and the cost of a download is constant. The coefficients for the formulas were first estimated by studying the actions of a typical implementation, and by roughly counting how many instructions each action would take. When it was hard to estimate the costs of actions, we ran measurement tests using simple emulation code. Finally, we experimentally validated the overall formulas against OpenNap performance, since OpenNap is an implementation of the chained architecture with the batch login policy.

For example, consider the cost of a query in batch mode. The cost of servicing a query consists of four main components: a startup cost that is constant across all queries, the cost of reading the inverted lists, the cost of reading the local results, and the cost of transferring remote results from

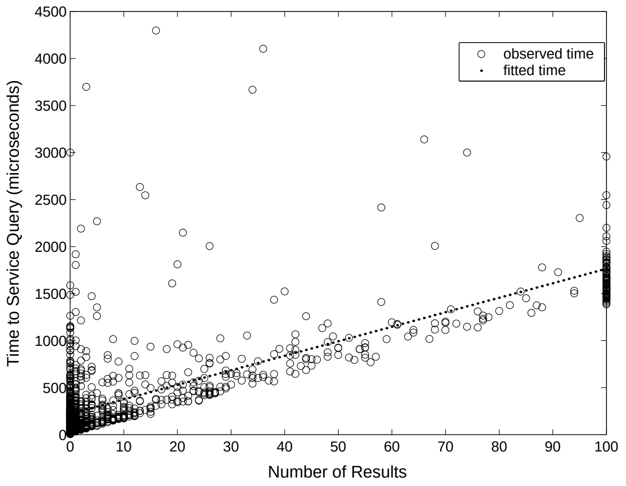

Figure 6: Time to Service a Query vs. Number of Results Returned

remote servers back to the local user. Because it is difficult to determine how much of a list will be read to answer a query, we modeled the number of list elements accessed as a linear function of the number of results returned. The cost for answering a query is then $c_1 \cdot \mathit{ExTotalResults} + c_2 \cdot \mathit{ExRemoteResults} + c_3 \cdot \mathit{ExServ}$, where c_3 is the startup cost, c_1 is the cost associated with finding one result, and c_2 is the cost associated with transferring one remote result back to the local user. By experimentally determining the cost of a list read, and finding base costs of in-memory transaction overhead and record reads from [14], we determined the coefficients seen in Table 6 for queries in batch mode. The first coefficient in the formula (4000) is the cost-per-result associated with processing the result, and the second coefficient (3778) is the cost-per-result associated with list access.

Figure 6 shows how well our model fits the observed data from the actual OpenNap server. The data we had available on query service cost did not include CPU costs incurred at remote servers, nor did it include the cost of transferring remote results. In effect, the data shows us the cost of a query when only one server is in the chain. Hence $\mathit{ExServ} = 1$, and $\mathit{ExRemoteResults} = 0$. Figure 6 shows the plot of *time* to service a query *for local results only*, versus the number of local results returned, as well as the analytic curve using the derived formula and CPU speed of the server. While the analytic curve is a rough approximation for a number of the datapoints, it does capture the general behavior of the majority of the datapoints. Later in Section 7, we show the effects of varying the cost-per-result coefficient (that is, the slope of the line in Figure 6) on system performance.

To calculate the cost of actions in incremental mode, we use the same formulas derived for batch mode, but incorporate the changes in the underlying action. Servicing a query in incremental mode has the same formula and coefficients as in batch mode, except that only *ActiveFrac* of the elements in the inverted lists pertain to files that are owned by currently active users, and all other elements cannot be used to answer the query. As a result, the cost of reading list elements per returned

Action	Formula
Batch Login	$42 + (75 + \mathit{WordsPerFile} \cdot \mathit{CharPerWord}) \cdot \mathit{FilesPerUser}$
Incr. Login	$42 + (46 + \mathit{WordsPerFile} \cdot \mathit{CharPerWord}) \cdot \mathit{FilesPerUser} \cdot \mathit{FracChange}$
Query	$\mathit{WordsPerQuery} \cdot \mathit{CharPerWord} + 100$
QueryResponse	$90 + \mathit{WordsPerFile} \cdot \mathit{CharPerWord}$
Download	$81 + \mathit{WordsPerFile} \cdot \mathit{CharPerWord}$

Table 7: Formulas for cost of actions in bytes

result needs to be divided by *ActiveFrac*.

The CPU cost of actions may also vary depending on architecture. In both the unchained and full replication architectures, the formula for batch and incremental query costs are exactly the same; however, $\mathit{ExServ} = 1$ and $\mathit{ExRemoteResults} = 0$ always. In the hash architecture, there is an additional cost of transferring every inverted list at a remote server to the local server. Hence, the coefficient describing the cost of list access per returned result is increased.

Network consumption. Table 7 lists formulas for the cost, in bytes, of the messages required for every action in the chained architecture. The coefficients come directly from the Napster network protocol, user-defined parameters, and a number of small estimates. The issue of network headers is discussed in Section 7.4.

For example, when logging on, a client sends a Login message with the user's personal information, an AddFile message for some or all files in the user's library, and possibly a few RemoveFile messages if the system is operating in incremental mode. The Napster protocol defines these three messages as:

- `Login` message format: `(MsgSize MsgType <username> <password> <port> ‘‘<version>’’ <link-speed>)`.
- `AddFile` message format: `(MsgSize MsgType "<filename>" <md5> <size> <bitrate> <frequency> <time>)`.
- `RemoveFile` message format: `(MsgSize MsgType <filename>)`.

Using the user-defined parameter values in Table 1, and estimating 10 characters in a user name and in a password, 7 characters to describe the file size, and 4 characters to describe the bitrate, frequency, and time of an MP3 file, the sizes of the Login, AddFile and RemoteFile messages come to 42 bytes, 125 bytes, and 67 bytes, respectively. When a client logs on in *batch* mode, a single Login message is sent from client to server, as well as *FilesPerUser* AddFile messages.[6] Average bandwidth consumption for payload data is $42 + 168 \cdot 125 = 21042$ bytes. When a client logs on in incremental mode, a single Login message is sent from client to server, as well as approximately $\mathit{FilesPerUser} \cdot \mathit{FracChange} \cdot 0.5$ AddFile messages, and the same number of RemoveFile messages. Average bandwidth consumption is therefore $42 + 168 \cdot 0.1 \cdot 0.5(125 + 67) = 1654.8$ bytes.

Network usage between servers varies depending on the architecture. For example, for login, the unchained architecture has no inter-server communication, so the required bandwidth is 0. Servers

[6] Yes, a separate message is sent to the server for each file in the user's library. This is inefficient, but it is the way the Napster protocol operates.

in the chained architecture send queries between servers, but not logins, so again, the bandwidth is 0. With the full replication architecture, all servers must see every Login, AddFile and RemoveFile message. If servers are connected on a LAN, then the data messages may be broadcast once. If the servers are connected on a WAN, however, then the local server must send the messages $NumServers - 1$ times. Similarly, in the hash architecture, if servers are connected via LAN, then the messages may be broadcast once. If the servers are connected via WAN, however, then the AddFile messages should only be sent to the appropriate servers. Please refer to Appendix A.4 for a description on calculating the expected number of servers each message must be sent to.

Modeling Overall Performance of an Architecture. After deriving formulas to describe the cost of each action, we can put everything together to determine how many users are supportable by the system. Our end goal is to determine a maximum value for *UsersPerServer*. To do this, we first calculate the maximum number of users supportable by each separate resource, assuming infinite resources of the other two types. Then, to find the overall *UsersPerServer*, we take the minimum across the three resources.

For example, let us find *UsersPerServer* for the chained system operating in batch mode, given default values for parameters listed in Tables 1 and 2. First, we will calculate the maximum number of supportable users for user-server communication over WAN. Because *UsersPerServer* is a function of *ExServ* and *ExTotalResults*, which are in return complex functions of *UsersPerServer*, we cannot calculate *UsersPerServer* directly. Instead, we use an iterative numeric procedure[7] where we guess two values for the parameter, calculate used bandwidth for each of these guesses, and interpolate a new value for *UsersPerServer* where zero bandwidth is unused, which is maximum capacity.

For example, let us suppose $UsersPerServer = 1000$. Using the formulas from Table 7, we know the cost of a login is 21042 bytes, the cost of a query is 112 bytes, and the cost of a download is 131 bytes. Then, for every query, $ExTotalResults = 22.22$ results are returned, using 3110 additional bytes for the QueryResponse messages. There are $QueryPerUserSec = .000833$ queries per user per second, $QueriesPerUserSec \ / \ QueryLoginRatio = .00184$ logins per user per second, and $QueriesPerUserSec \ / \ QueryDownloadRatio = .00166$ downloads per user per second. Total bandwidth per user is therefore 333.6 bits per user per second, meaning unused bandwidth is 7.6664 Mb/s. We then use our iterative procedure to find the *UsersPerServer* that gives us zero unused bandwidth, yielding 21851 users, the maximum number of supportable users for user-server communication over WAN.

Repeating the same process for inter-server communication and CPU resources, we find that the maximum number of supportable users for inter-server communication is 996199 users per server, and for CPU is 16382 users per server. The minimum of these values is 16382 users per server, meaning our system can support at most 16382 users per server given all other parameters, and that CPU is the bottleneck in this case.

[7] The procedure we use is the secant method for zero-finding.

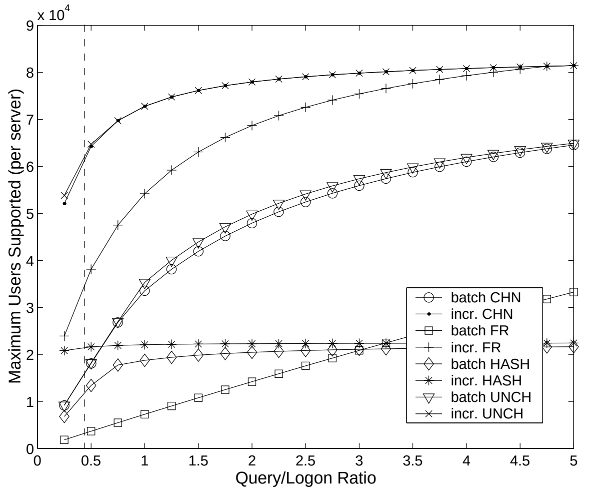

Figure 7: Overall Performance of Strategies vs. *QueryLoginRatio*

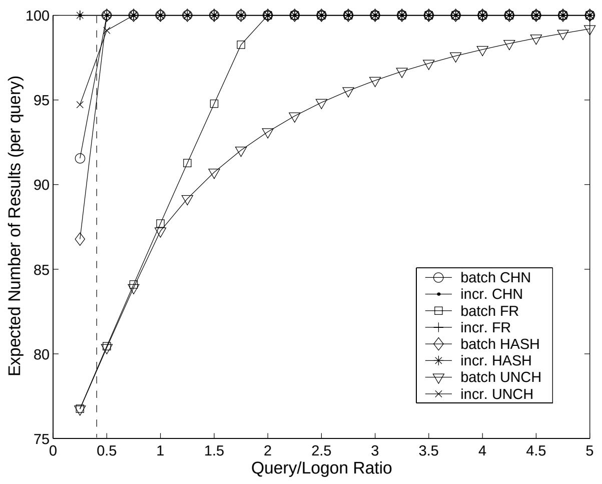

Figure 8: Expected Number of Results

7 Experiments

In this section, we present the results of our performance studies, discussing the behavior of the architectures and login policies as certain key parameters are varied, and highlighting the tradeoffs between architectures in different scenarios. We begin by studying music-sharing systems today (Section 7.1). Next, we consider how such systems might perform in the future when user and system characteristics have changed (Section 7.2). Finally, in Section 7.3 we switch focus to other domains and evaluate the performance of systems that may have very different query and indexing characteristics from music-sharing applications.

Throughout the performance comparisons, the metric of performance is the maximum number of users each server can support. Hence, we concern ourselves with throughput, and not response time. Unless otherwise specified, default values for parameters (see Tables 1, 2, and 3) were used during evaluation. For brevity, we will refer to the chained architecture as CHN, the full replication architecture as FR, the hash architecture has HASH, and the unchained architecture as UNCH. Because each architecture can be implemented using one of two login policies, we refer to the combination of a particular architecture and policy as a “strategy”. For example, “batch CHN” is the strategy where the chained architecture is implemented using the batch login policy. There are a total of 8 strategies.

7.1 Music-Sharing Today

We begin by evaluating performance of systems with behavior similar to that of music-sharing systems today (e.g., Napster, OpenNap), described by default user and system parameter values listed in Tables 1 and 2. Figure 7 shows the overall performance of the strategies over various values of *QueryLoginRatio*. For example, at *QueryLoginRatio* = 1, incremental FR can support 54203 users per server, whereas batch FR can only support 7281 users per server. The dashed line represents the experimentally derived value of *QueryLoginRatio* for OpenNap. Figure 8 shows the expected number of results per query for each of the strategies shown in Figure 7, assuming that *MaxResults* = 100. As *QueryLoginRatio* increases, the number of logins per second decreases, meaning more users can be supported by the system, thereby making more files available to be searched. As a result, when *QueryLoginRatio* increases, the expected number of results increases as well.

As seen in Figure 7, the incremental CHN and UNCH strategies have the best performance. Both strategies are bound by user-server communication over WAN. The FR strategies are bound by CPU due to expensive logins. However, as *QueryLoginRatio* increases, logins become less of a factor in overall performance, and we see how incremental FR performance improves until *QueryLoginRatio* = 5, where WAN becomes the bottleneck and the strategy has equal performance to incremental CHN and UNCH. Finally, HASH is bound by inter-server communication over LAN because of the large lists that must be transferred during queries. In many of the scenarios we study in later figures, we will see that as a general rule, the FR bottleneck is CPU when *QueryLoginRatio* is low, and the HASH bottleneck is inter-server communication.

We can also see in Figure 7 that user-server communication for incremental strategies, as seen in the performance of incremental CHN, is better than batch user-server communication performance, as seen in the performance of batch CHN. In fact, incremental user-server communication performance is always better than batch performance, because while queries consume the same bandwidth in both policies, logins use significantly less bandwidth in incremental than in batch. Furthermore, since all architectures adhere to the same message protocol for user-server communication, all architectures have roughly the same user-server communication performance when operating under the same login policy.

Now, let us consider only the *batch* login policy, which is the policy we believe to be closest to Napster. The top two batch strategies, CHN and UNCH, have roughly the same performance. However, batch UNCH returns substantially fewer results per query than batch CHN. Figure 8 shows the expected number of results per query, assuming that *MaxResults* = 100. As *QueryLoginRatio* increases, the number of logins per second decreases. As a result, more users can be supported by the system, meaning more files are available to be searched, and more results are returned. Most strategies reach the maximum quickly, but batch UNCH has a substantially lower expected number of results, and "catches up" as *QueryLoginRatio* increases. By the time expected number of results reaches *MaxResults*, batch UNCH has the same exact performance as batch CHN. Hence, batch UNCH only has better performance than batch CHN when it returns fewer results; furthermore, the performance difference is very small, while the expected number of results difference can be quite large. If the current parameters do indeed describe Napster's systems, then the benefits of UNCH in capacity may not outweigh the disadvantage of fewer results, and batch CHN would be a better strategy than the current batch UNCH strategy.

In summary, from this experiment we can make several important conclusions:

- Incremental strategies outperform their batch counterparts. In particular, incremental CHN and UNCH have the best performance and are recommended in this scenario. Currently, the incremental policy is not used by music-sharing systems, but it should clearly be considered (see end of section for memory requirement considerations).

- In a network-bound system, incremental strategies will always outperform batch because while queries consume the same bandwidth in both policies, logins use significantly less bandwidth in incremental than in batch.
- Batch UNCH is the strategy that most closely describes Napster's architecture. As seen in the figures, surprisingly, adding chaining to the servers (switching from UNCH to CHN) does not affect performance by much, but returns significantly more results. Assuming the cost of maintaining a LAN between the servers is acceptable, batch CHN is clearly recommended over batch UNCH.
- Most policies are very sensitive to *QueryLoginRatio* near our measured value of 0.45. Small changes in *QueryLoginRatio* can significantly increase or reduce the maximum number of users supported by the system, thereby making capacity planning difficult. This sensitivity is especially important to consider if large increases in *QueryLoginRatio* are expected in the future when user network connections become more stable (see Section 7.2).
- As a general rule, FR is CPU-bound due to expensive logins when *QueryLoginRatio* is small, HASH is bound by inter-server communication because of the large lists that must be transferred during queries, and CHN and UNCH are bound by user-server communication.

From this point on, we will no longer include the unchained architecture in our comparisons because the tradeoff is always the same: better performance, but fewer results per query.

Batch versus Incremental CPU Performance. In Figure 7, we saw that incremental strategies always have better user-server communication performance than batch strategies. Now we would like to see how they compare in terms of CPU performance. From the formulas in Table 6, the incremental policy clearly has better login performance, especially as *FracChange* decreases. However, incremental also has a much worse query performance because the server must filter out inactive file information. Therefore, which strategy is better should depend on the ratio of queries to logins that the server must handle. In Figure 9, we see the CPU performance (that is, maximum users supported by the CPU, assuming network communication is not the bottleneck) of each strategy as *QueryLoginRatio* varies in value. As expected:

- In a CPU bound system, incremental strategies only outperform batch when *QueryLoginRatio* is small, because a small ratio of queries to logins is in their favor. Note that in the previous experiment, the system was partially network bound, so this rule does not apply.
- Batch strategies are recommended over incremental strategies in a CPU bound system where *QueryLoginRatio* is large. As seen in the figure, the CPU performance of all batch strategies improves as *QueryLoginRatio* increases, until it surpasses the performance of the incremental strategies.

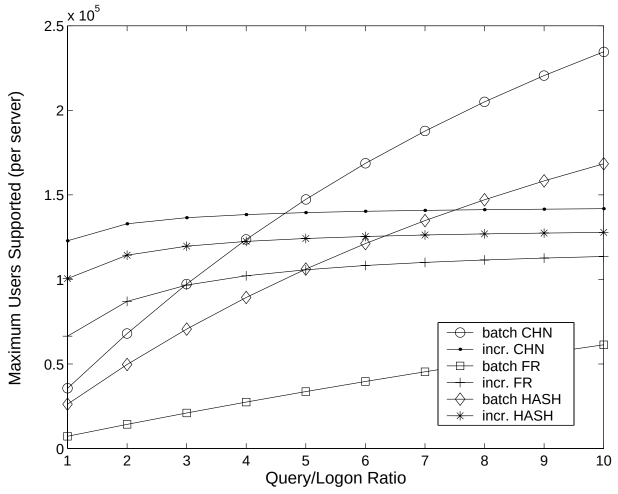

Figure 9: CPU Performance of Login Policies vs. *QueryLoginRatio*

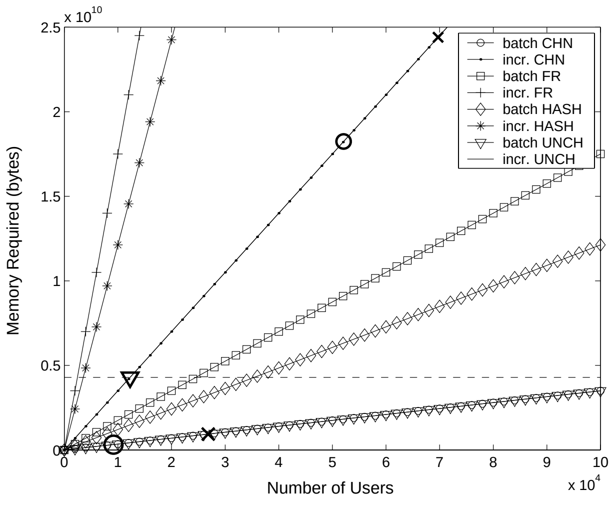

Figure 10: Memory Requirements

Note that the balance between incremental and batch depends on *FracChange* and *ActiveFrac* as well. If *FracChange* is small and/or *ActiveFrac* is large, then the balance favors the incremental strategy. The effect of these two parameters is discussed in further detail in Section 7.4.

Memory Requirements Thus far, we have evaluated the strategies assuming that there was enough memory to hold whatever indexes a server needed. We will now take a closer look at the memory requirements of each strategy.

Figure 10 shows the memory requirement in bytes of the various strategies, as a function of the number of users. Please refer to Appendix C for a description on how memory usage is calculated. Here, we assume *ActiveFrac* = .1, to keep all architectures within roughly the same scale. Clearly, the batch strategies are far more scalable than the incremental strategies. For example, when there are 10000 users in the system, batch CHN requires .35 GB of memory, while incremental CHN requires 10 times that amount. Also, CHN requires the least amount of memory, while FR requires the most. However, it is important to note that memory requirement is a function of several parameters, most importantly *NumServers* and *ActiveFrac*. As *NumServers* decreases, FR requires proportionally less memory. On the flip side, as *NumServers* increases, FR also requires proportionally more memory. Likewise, incremental strategies require 1/*ActiveFrac* as much memory as batch. As connections become more stable and *ActiveFrac* increases, memory required by incremental strategies will decrease inverse proportionally, until it is much more comparable to batch memory requirements than it currently is. Furthermore, as memory becomes cheaper and 64-bit architectures becomes widespread, memory limitations will become much less of an issue than it is now.

Today, it is likely that system administrators will limit the memory available on each server. By imposing a limit, several new tradeoffs come into play. For example, suppose a 4GB memory limit is imposed on each server, shown by the dashed line in Figure 10. Now, consider a Napster scenario where $r = 4$, $\lambda_f = 100$, and $\mathit{ActiveFrac} = .1$. Say we determine that $\mathit{QueryLoginRatio} = .75$. Our model predicts that the maximum number of users supported by batch CHN is 26828, and by incremental CHN is 69708. The memory required by these two strategies is shown in Figure 10 by the large 'x' marks. While incremental CHN can support over twice as many users as batch CHN, it also requires very large amounts of memory – far beyond the 4GB limit. If we use the incremental CHN strategy with the 4GB limit, then our system can only supported 12268 users per server, shown as a large triangle in Figure 10, which is fewer than the users supported by batch CHN. Hence, batch CHN is the preferred architecture for this scenario.

However, let us now suppose *QueryLoginRatio* is .25. Then, the maximum number of users supported by batch CHN is 9190, and by incremental CHN is 52088. The memory required by these two strategies is shown in Figure 10 by the large 'o' marks. Again, the amount of memory required by incremental CHN is far too large for our limit. However, looking at incremental CHN performance at the 4 GB limit, we find that the 12268 users supported by incremental CHN is greater than the 9190 users supported by batch CHN. Hence, incremental CHN is still the better choice, because within the limit, it still has better performance than batch CHN.

7.2 Music-Sharing in the Future

In the future, a number of user and system characteristics can change, thereby affecting the performance of music-sharing systems. Here we consider three types of changes: change in network stability, change in user query behavior, and change in system hardware capacity.

Changes in Network Stability. As mentioned in Section 5, the experimentally derived value for *QueryLoginRatio* is surprisingly low, and this is most likely due to the fact that most users today have unstable network connections through dialup modems. In the future, we would expect more users to have stable, high bandwidth connections from home through DSL or cable modem. How would system performance change given this trend in network stability? First, if connections are stable and users can remain online for longer periods of time, then we would expect *QueryLoginRatio*, the number of queries a user submits per session, to rise from its current value of 0.45. $\mathit{QueryLoginRatio} = 0.45$ means that the average user submits 1 query every other time s/he logs on, which is very low. We would expect that in the future, *QueryLoginRatio* is at least an order of magnitude larger, so as a future estimate we will use $\mathit{QueryLoginRatio} = 10$.

Second, if users remain online for longer periods of time, then at any given moment, a higher fraction of total users are logged in. The current value for $\mathit{ActiveFrac} = 0.05$, meaning 1 out of 20 users are logged in at any given time. Though *ActiveFrac* will increase, it probably will not increase as much as *QueryLoginRatio*, so as a future estimate, we will use $\mathit{ActiveFrac} = 0.25$, meaning 1 out of 4 users are logged in at any given time.

Figure 11 shows the performance of the strategies given the default parameter values listed in Tables 1 and 2, but with $\mathit{QueryLoginRatio} = 10$ and $\mathit{ActiveFrac} = 0.25$. We show performance as a function of the number of servers in the system, in order to compare the scalability of the

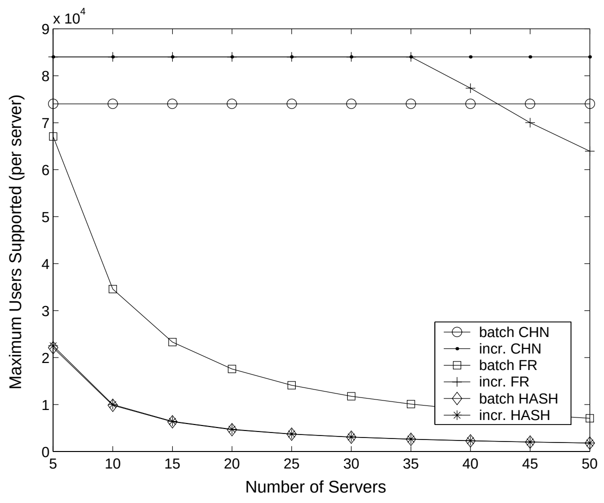

Figure 11: Overall Performance of Strategies vs. Number of Servers

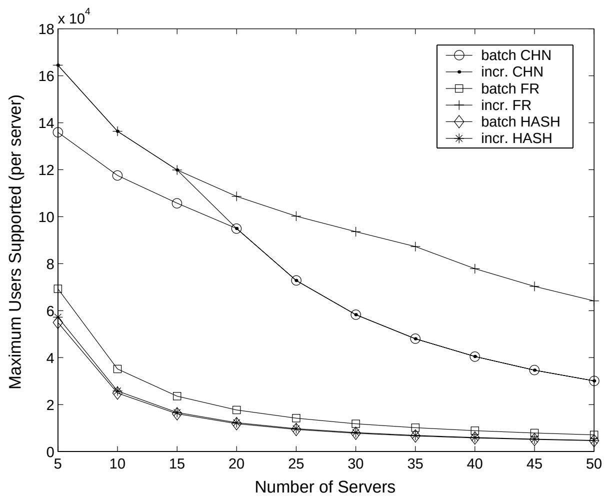

Figure 12: Overall Performance of Strategies vs. Number of Servers

strategies. Note that the vertical axis shows the maximum number of users *per server*. A scalable strategy should exhibit a fairly flat curve in this figure, because as more servers are added to the system, the capacity at each server remains near-constant and the system will have best-case linear scalability. The lessons in scalability we gather from this experiment are:

- Incremental CHN continues to be the best performing strategy, and one of the most scalable as well. CHN's scalability is due to the fact that most operations involve only the local server. Logins and downloads are handled at the local server only, and because *ExServ* is very low with $\lambda = 100$ and $r = 4$, queries are mostly handled at the local server as well. In the next experiment, we see how CHN scales poorly when *ExServ* is high.
- FR is not scalable because the cost of downloads and logins increase in proportion to the number of servers in the system. Since the incremental policy has better login performance than batch, incremental FR has good performance until the number of servers becomes large ($\tilde{}$35). However, batch FR is always unscalable because of the CPU cost of logins. If CPU speeds increase such that CPU is no longer the bottleneck, batch and incremental FR would perform and scale as well as batch and incremental CHN, respectively.
- HASH is not scalable in this scenario because all operations become more expensive as the number of servers increase, in terms of inter-server communication. However, we show next that if network bandwidth increases such that it is no longer the bottleneck, HASH can have the best performance and scalability. In addition, we show that when the number of words per query is low, HASH is by far the most scalable.

Changes in Query Model. Future changes in query model parameters will also affect the performance of strategies. For example, suppose user interests become more diverse. In this case, query popularity and selection power are less skewed, meaning λ and r are larger. In addition, diverse interests translate into fewer expected results in the query model, meaning users will likely submit more queries before finding the files they wish to download. In this case, *QueryDownloadRatio*, the number of queries submitted before a single download, will increase as well.

Figure 12 shows us performance of strategies when user interests are diverse, that is, where $r = 16$, $\lambda = 1000$, and *QueryDownloadRatio* $= 2$. With Napster, $\lambda = 100$ and $r = 4$, meaning the 100th query had the mean selection power, and the 400th query had the mean popularity. In this scenario, both distributions are more evenly distributed, so that only the 1000th query has the mean selection power, and the 16000th query has the mean popularity. With OpenNap, *QueryDownloadRatio* $= 0.5$, meaning a user downloaded 2 files for every query submitted. Now, with fewer expected results per query, we estimate that users will only download a file with every other query submitted, meaning *QueryDownloadRatio* $= 2$. In this figure, we also assume that network connections are stable as in the previous example, meaning *QueryLoginRatio* $= 10$ and *ActiveFrac* $= 0.25$.

We see in this scenario that incremental FR has the best performance. Although FR is still bound by CPU, now CHN is bound by inter-server communication, because diverse interests mean fewer results, and fewer results means higher *ExServ* for CHN. CHN does not scale well with high *ExServ*, because many query and query result messages must then be exchanged between servers. FR and HASH, however, are relatively unaffected by changes in r and λ, because *ExServ* in these strategies do not depend on these parameters.

Future changes to the query model are difficult to predict; however, from the above experiment and other experiments not shown, we observe several general trends:

- Changes in r have a much larger effect on system performance than changes in λ. Changes in λ are generally only noticeable when r remains the same.
- If r and/or λ decreases, then the distributions are more skewed (e.g., "narrow interests"), and *ExServ* remains small. As a result, CHN queries are still scalable, and relative performance of the strategies is unaffected by the changes. Incremental CHN continues to have the best performance and scalability.
- If r and/or λ increases, then the distributions are more evenly distributed (e.g., "diverse interests"), and *ExServ* increases. As a result, CHN queries are unscalable, and incremental FR has the best performance and scalability.

Changes in Hardware Capacity. In the future, we may also expect changes in hardware capacity. For example, suppose network bandwidth for both WAN and LAN increases such that network is no longer the bottleneck for any of the strategies. Figure 13 shows the same scenario illustrated by Figure 12, but where CPU is the limiting resource for all strategies. We see from this figure that HASH is now the most scalable of the strategies. After the system scales beyond 10 servers, both batch and incremental HASH curves become rather flat, and after 20 servers, incremental HASH has the best performance. HASH is scalable in regards to CPU because beyond a certain point, both login and query performance are almost unaffected by the number of servers in the system. For logins, every song is replicated at most *WordsPerFile* times, if every word in

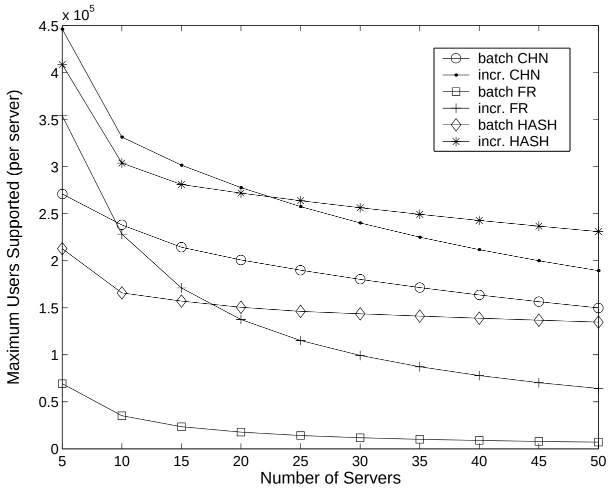

Figure 13: Overall Performance of Strategies vs. Number of Servers

the metadata hashes to a different server. Since *WordsPerFile* = 10, after *NumServers* = 10, the degree of replication barely increases as the number of servers increase. Login notices are still sent to every server, but the size of a notice is very small compared to library metadata. For queries, at most *WordsPerQuery* servers are involved in a query. Hence, beyond *NumServers* = 2.4, the number of servers involved in a query remains basically constant. The other strategies are not as scalable as HASH for reasons mentioned before. CHN is very scalable in terms of logins, but not queries if *ExServ* is high (e.g., if r is high). As the system scales up, more servers are involved in satisfying a query, so performance per server decreases. FR is very scalable in queries but not logins, so as the number of servers increases, logins take a proportionally larger slice of the CPU. From the above experiment and experiments not shown here, we find that:

- If the CPU only becomes faster, then incremental and batch FR will improve in performance, but the other strategies will not.
- If the network becomes faster, then both CHN and HASH will improve in performance, but HASH will improve more than CHN.

In general, changes in hardware capacity will only improve the performance of those strategies that currently experience a bottleneck with that resource.

7.3 Beyond Napster: Systems in Other Domains

Thus far, we have considered systems that are similar to Napster, with the simple query and indexing scheme described in Section 3, and where the distributions in the query model are assumed in Section 4 to be exponential. In this section, we study the performance of systems that have very different query and system behavior than current music-sharing systems. Due to lack of space, we cannot cover all possible variations of systems. Instead, we describe the types of systems that can be evaluated well with our models, and show a few examples.

	$f(i)$	$g(i)$	Parameters	Correlation	Approx. Pos. Distributions
(a)	$rand(i)/\lambda_f$	$\frac{1}{\lambda_g}e^{-\frac{i}{\lambda_g}}$	$\lambda_f = 100; \lambda_g = 400$	None	$\lambda_f = 100, r = 1$
(b)	$rand(i)/\lambda_f$	$\frac{1}{\lambda_g}e^{-\frac{i}{\lambda_g}}$	$\lambda_f = 1000; \lambda_g = 700$	None	$\lambda_f = 1000, r = 1$
(c)	$\begin{cases} \frac{1}{\lambda_f}e^{-\frac{10000-i}{\lambda_f}} & \text{i} < 10000 \\ \frac{1}{\lambda_f}e^{-\frac{i}{\lambda_f}} & otherwise \end{cases}$	$\frac{1}{\lambda_g}e^{-\frac{i}{\lambda_g}}$	$\lambda_f = 100; \lambda_g = 400$	Negative	$\lambda_f = 100, r = 32$

Table 8: Potential Distributions for f and g

Query Model. In Section 4, we describe a query model where, given distributions for query frequency and query selection power, we can calculate the expected number of results for a query, and the expected number of servers needed to satisfy the query. The model itself is completely general in that it makes no assumptions about the type of distributions used for f and g; however, for the purposes of modeling music-sharing systems and validating the model, we made the temporary assumption that f and g are exponential distributions. Implied by this assumption is the additional assumption that f and g are *positively correlated* – that is, the more popular or frequent and query is, the greater the selection power it has (i.e. the larger the result set).

While we believe that our "music" model can be used in many other domains, one can construct examples where the f and g distributions have different properties. For example, if the system supported complex, expressive relational queries, an obscure query such as `select * from Product where price > 0` would return as many results as a common query such as `select * from Product`. In this case, there is very little, if any, correlation between the popularity of a query and the size of its result set. We say that such systems have *no correlation* distributions. Another example might be an "archive-driven" system, where users provide old, archived files online but keep their current, relevant information offline. This scenario might occur in the business domain where companies do not want their current data available to the public or their competitors, but might be willing to provide outdated information for research purposes. Because archives hold historical data, popular queries which refer to recent times return less data, whereas rare queries which refer to the past will return more. In this case, there is a *negative correlation* between query popularity and selection power.

We now show how distributions that fall into the negative and no correlation categories can be approximated quite well by distributions in the positive correlation category. Table 8 shows three representative f and g distribution pairs from the various categories. The first pair represents the no correlation category. Distribution g is still exponential, since we know it must be a monotonically decreasing function that sums to 1, and exponential distributions are common in real-life systems. Distribution f is a random function, producing random probabilities falling within the range $[0, \frac{1}{\lambda_f}]$. We choose this range for comparison purposes, since a positively correlated model with the same parameters would have that same range for f. The second pair is similar to the first, except that parameters λ_f and λ_g are different. The third pair is representative of the class of negative correlation distributions. For the top 10000 most popular queries (which accounts for almost 100%

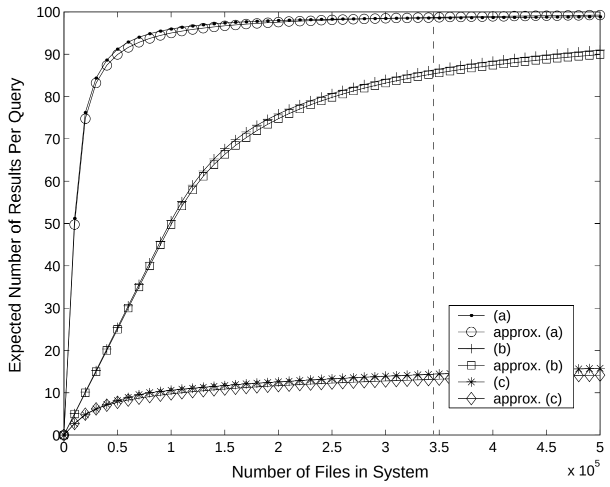

Figure 14: Expected Number of Results vs. Number of Files

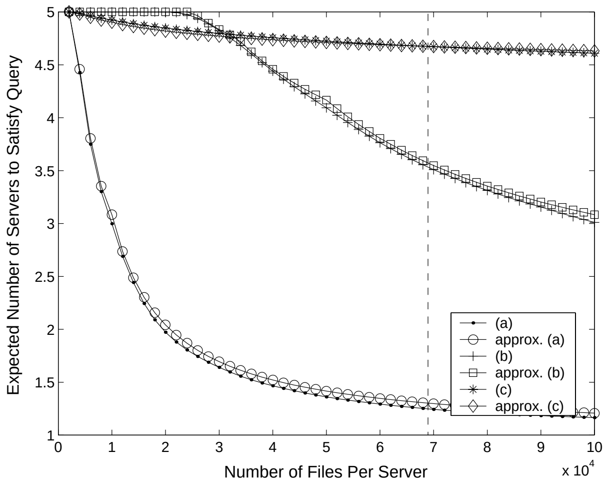

Figure 15: Expected Number of Servers vs. Number of Files Per Server

of all query occurrences), the more popular the query, the smaller the result set. After the first 10000 queries, we allow the less popular queries to have smaller result sizes in order to account for misspellings, searches for non-existent data, etc. Again, g is exponential, for the same reasons as the first two pairs. Table 8 also lists, for each pair, the parameters for a positive correlation distribution that approximates the behavior of the actual negative or no correlation pair.

Figures 14 and 15 shows *ExResults* and *ExServ* for each of the representative distributions. *ExResults* is shown in Figure 14 as a function of the *total* number of files in the system. In Figure 15 *ExServ* is shown as a function of the number of files *per server*, and there are assumed to be 5 servers in the system. The dashed lines in both figures show values from the OpenNap system. In addition to showing the curves from the actual negative and no correlation distributions, Figures 14 and 15 also show the *approximation curves* derived from the positive correlation approximation distributions listed in Table 8. As we can see, the approximations fit the actual curves for *ExResults* and *ExServ* very well. Since the sole purpose of the query model is to calculate values for *ExServ* and *ExResults* given distributions for f and g, we know that overall performance of a system with the negative or no correlation distributions will be closely matched by a system with the corresponding positive correlation approximation distributions.

Though we can only show a few examples of negative and no correlation distributions here, additional experimentation has shown that any no-correlation distribution like the ones in Table 8, where $\lambda_f = c_f$ and $\lambda_g = c_g$, can be almost exactly approximated by a positive correlation distribution where $\lambda_f = c_f$ and $r = 1$. In addition, though there is no nice formula to describe the relationship, any negative correlation distribution model can be fairly well approximated with a positive correlation distribution where r is very large. This makes sense because positive correlation models with large r have low *ExResults* and therefore high *ExServ*, while negative correlation distributions result in low *ExResults* (and therefore high *ExServ*) as well, since the queries that occur most frequently all return small result sets.

Now that we have shown that distributions with all three types of correlation can be approximated quite closely by positive correlation distributions, by studying the performance of systems with a wide range of positive correlation distributions, we can understand the behavior of systems with negative and no correlation distributions as well.

Figure 16 shows CPU performance of each strategy as r is varied. For comparison, the r values derived for the OpenNap system, and for the approximations of the negative and no correlation curves, are shown by dotted lines in the figure. First, we notice that increasing r has a much larger effect on the incremental strategies than on batch. For example, at $r = .25$, batch CHN can support 145000 users per server, while incremental CHN can support 110000 users. By the time r reaches 32, however, incremental CHN has seen a 250% improvement in performance, while batch CHN has remained relatively constant. The cause behind this difference is again due to the fact that incremental has poor query performance and good login performance, while batch has poor login performance but good query performance. When r is small, *ExTotalResults* is large, and is limited by *MaxResults*. Queries are thus as expensive as possible, and batch performs better than incremental. In fact, when r increases to 4, the performance of each strategy barely changes because *ExTotalResults* is still equal to *MaxResults*. (However, *ExServ* is increasing. In reality, performance is decreasing very slightly from $r = .25$ to $r = 4$ because of the the extra startup cost at the additional servers, but this decrease is difficult to see in the figure). However, once r increases beyond 4, *ExTotalResults* begins to decrease, first rapidly, then slowly. As a result, the incremental strategies, which up to a certain point were weighed down by slow queries, suddenly improve in performance. The batch strategies, however, which are weighed down by slow logins, barely improve.

In summary, we can make several observations about the performance of strategies in different query models:

- With negative correlation distributions, or positive correlation distributions with high r, incremental strategies are recommended over batch. CPU performance of incremental strategies is usually bound by expensive queries (see Section 7.1), but with negative correlation or high r, the expected number of results is very low, thereby making queries inexpensive.
- With no correlation distributions, or positive correlation with low r, batch strategies outperform incremental (except for FR). This is because when r is low, the expected number of results is very high, thereby making queries expensive and incremental performance poor. Again, batch FR performs so poorly because of expensive logins.
- For the same reason that incremental shows greater improvement than batch, CHN and HASH show greater improvement than FR as r increases. FR, which has poor login performance but excellent query performance, is least affected by a decrease in *ExResults* caused by an increase in r. Hence, its performance improves the least as r increases.

Performance Model. Unlike the query model described in Section 4, the performance model described in Section 6 does make assumptions about the query and indexing scheme used in the

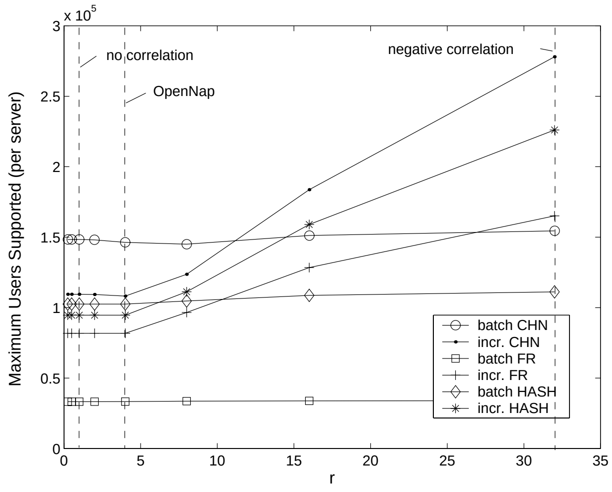

Figure 16: CPU Performance of Strategies vs. Query Model Parameter r

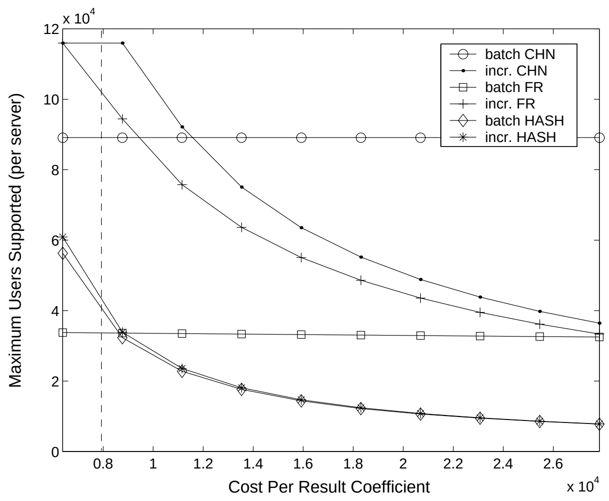

Figure 17: Overall Performance of Strategies vs. List Access Coefficient

system. These assumptions are implied by the *structure* of the cost formulas listed in Tables 6 and 7. For example, Table 6 lists that the CPU cost of a query has the form $c_1 \cdot x + c_2 \cdot y + c_3 \cdot z$, where x is the expected number of results, y is the expected number of remote results, z is the expected number of servers needed to satisfy the query. This formula reflects the assumption that *ExTotalResults*, *ExRemoteResults* and *ExServ* have linear effects on query cost, and that they are the only variables affecting query cost. Clearly, the actual cost of a query in any system would be a much more complex expression involving a number of variables; however, as long as the cost of a query can be approximated well within these assumptions, we can model the system simply by setting coefficients to values appropriate for that system (which we show in the next example). It is only if the cost of processing a query can not be approximated well within these assumptions that the system cannot be analyzed well within our performance model.

For systems that can be described by our performance model, performance depends heavily on the coefficients used in the cost formulas. For example, in Section 6 we described how we derived the CPU cost-per-result coefficient for queries in the OpenNap system. However, we can easily imagine how other systems could have very different costs per result. For example, suppose queries consist of a simple key or ID, and the indices are simple hash tables. When a user submits a query, the server simply probes into the hash table with the given ID. In this case, searches would be very inexpensive in terms of CPU cycles – possibly on the order of hundreds of cycles. On the other extreme, suppose the system holds an image database, and queries consist of low resolution images. The system is meant to retrieve all images that are similar to the query image by some comparison technique and criteria. Processing a query in this system would be very expensive because for every result, the server would have to run an expensive image comparison algorithm, and furthermore the cost for all non-matching comparisons would be amortized over the cost for each match. In this case, cost-per-result could be on the order of millions of cycles.

Figure 17 shows the effect that varying the cost-per-result coefficient has on overall performance.

The coefficient derived in Table 6 for the OpenNap system is 7778, marked by a dotted line in the figure. A higher value for the coefficient, for example, 15556, means that twice as many cycles are required to both search for and process a single result.

Incremental strategies are most negatively affected by increases in the coefficient, since the negative effect on query performance is magnified by $1/\mathit{ActiveFrac}$ (see Table 6). Batch CHN and incremental CHN both start off bound by user-server communication, but as the value for the coefficient increases beyond 9000, incremental CHN becomes CPU-bound, and performance degrades rapidly. Hence for small values of the coefficient, incremental CHN has the best performance, since incremental user-server communication is better than that of batch. However, batch CHN has the best performance when the coefficient is large, because CPU performance is not drastically affected as it is with incremental, and does not become the bottleneck in the range shown. As for the other architectures, we see that once again, FR is CPU-bound and HASH is inter-server communication bound.

In summary, from this experiment we find that:

- For systems with a small cost-per-result coefficient (e.g., OpenNap, hash-lookup systems), incremental strategies are recommended over batch, particularly incremental CHN.
- For systems with large cost-per-result (e.g., image databases), batch CHN is the recommended strategy. Batch performs better when cost-per-result is high because incremental strategies suffer from poor CPU query performance (see Section 7.1).
- Batch CHN is also recommended for systems where cost-per-result is unknown or highly variable, since the performance of batch CHN is steady, while even a small increase in cost-per-result causes the performance of incremental CHN to decreases dramatically.

If necessary, all cost coefficients can be varied in the same way in order to view how systems with different query and indexing characteristics perform.

7.4 Miscellaneous Experiments

In this section, we present additional experiments help us to better understand the strengths and weaknesses of each strategy.

Performance Impact of MaxResults. Currently, file-sharing systems in the music industry set a rather low limit on the number of results returned for a query. Setting a limit is good practice because queries are narrow, and limiting the number results can save bandwidth while not preventing the user from finding what she is looking for. For example, searching for "erasure always" on Napster will return 100 copies of that same song. However, file-sharing systems in other domains may not have the same kind of narrow search. Instead, searches are on a topic, and the user wants all results returned. For example, if the domain were photography, and a user wished to search for photographs of "yellow cars", every photograph returned would be unique, and the user might want to see every one before deciding which ones to download. In such a scenario, *MaxResults* is effectively set at ∞.

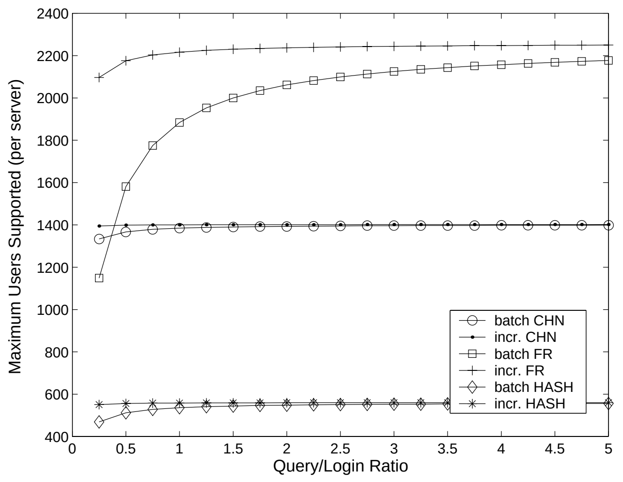

Figure 18: Overall Performance of Strategies with MaxResults = ∞

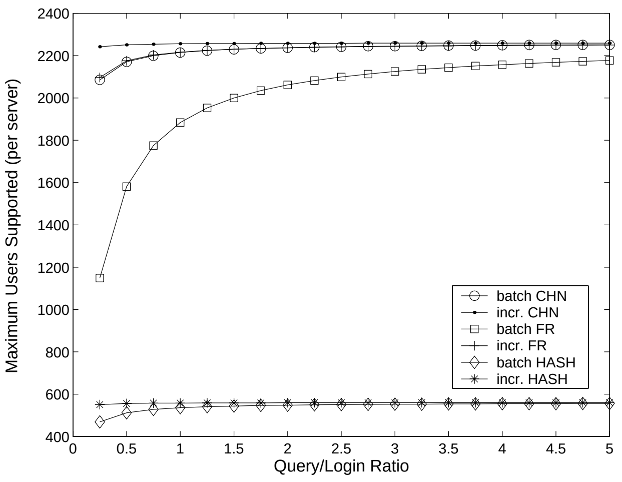

Figure 19: Overall Performance of Strategies with MaxResults = ∞ and Chained-Direct

When $MaxResults = \infty$, inter-server bandwidth can become a problem. For CHN, remote results are sent from remote server to local server before being returned to the user. For HASH, inverted lists must be sent between servers. For FR, however, queries do not consume any inter-server bandwidth because every local server contains a complete index. Hence, we expect FR to perform the best in this scenario. Figure 18 shows the performance of architectures when $MaxResults = \infty$, and inter-server communication is over WAN, along with user-server communication.

In this scenario, every architecture is bound by the combined network bandwidth. As expected, we see in the figure that FR performs the best, with the incremental FR strategy having over 55% better performance than both CHN strategies, and batch FR having nearly as good performance as incremental when *QueryLoginRatio* increases (because the expensive logins become less frequent). Again, HASH performs the worst in network bandwidth.

One might argue that CHN could save much bandwidth by having remote servers return results directly to the user, rather than routing them through the local server. The cost of such an implementation, which we will call "chained-direct", would be added complexity to the client. Figure 19 shows performance of the strategies in the same scenario, but where the chained-direct implementation is used for CHN. We can see that chained-direct has only negligibly better performance than incremental FR, and again, batch FR has close to as good performance with larger *QueryLoginRatio*. The reasons chained-direct performance is not significantly better than FR are (1) queries must still be sent from server to server, even if results are not, and (2) with a large enough *QueryLoginRatio*, the extra inter-server communication used by FR's login messages is s-mall in comparison to all the other network costs (e.g., user-server communication). In fact, FR can still perform better than chained-direct if *QueryLoginRatio* is quite large, and *FracChange* is small.

Another benefit of FR is query response time. Although we do not study this aspect of performance, FR would clearly have the best response time since all processing occurs at the local server

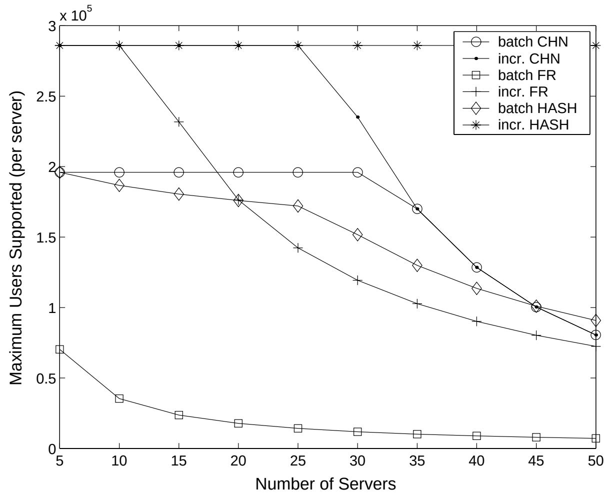

Figure 20: CPU Performance of Strategies vs. Number of Servers in System

only. All other architectures require relatively slow inter-server communication. Of course, FR also has the slowest login response time. However, this is generally considered to be a less important performance metric from the user's point of view, especially with higher *QueryLoginRatio*.

In summary, from these experiments we learn that:

- When remote query result sets are large, either in number of results or in size in bytes, then FR is recommended because it is least affected by query bandwidth usage.
- If, in the CHN architecture, we send remote results directly from a server to a remote user, then CHN and FR have comparable performance, but the CHN client contains additional complexity.
- Although not quantitatively considered in this paper, query response time is an important performance metric, and FR has the best performance in this respect.

Other Sensitivity Analysis. In Section 5 we promised sensitivity analysis on *ActiveFrac* and *FracChange*, as these values could not be determined by the OpenNap experiments. The effects of the two parameters were summarized in the discussion of Figure 9. *ActiveFrac* and *FracChange* affect only the incremental strategy. *ActiveFrac* affects only queries, and *FracChange* affects only logins. As *ActiveFrac* increases, a larger fraction of the entire user population is online at any given time. Hence, fewer useless list accesses are made for every query, decreasing query costs. When *ActiveFrac* is low then queries are more expensive, so query-intensive architectures such as CHN and HASH perform best. As *FracChange* decreases, fewer offline changes are made to libraries, and hence, less work and bandwidth is needed for logins. When *FracChange* is low then logins are expensive, so login-intensive architectures such as FR perform best.

Single-Word Query Systems. In all scenarios considered so far where inter-server bandwidth is limited, HASH has very poor performance because of the large amount of list data transferred during queries. The amount of data transferred is a function of how many terms are in the query;

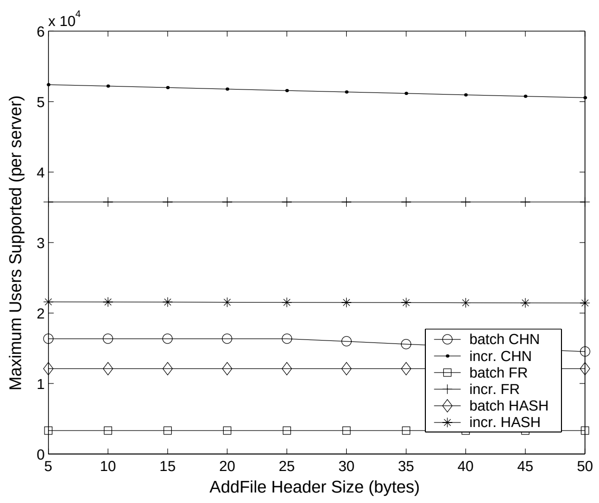

Figure 21: Overall Performance of Strategies vs. AddFile Packet Header Size

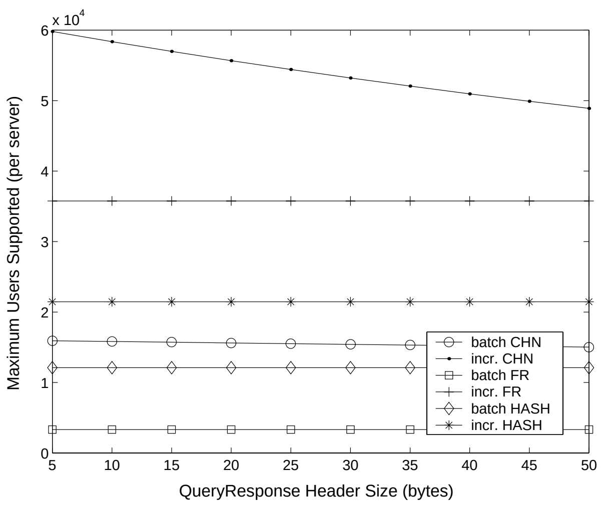

Figure 22: Overall Performance of Strategies vs. QueryResponse Packet Header Size

in the worst case, *WordsPerQuery* - 1 inverted lists are transferred over the inter-server connection per query. In some systems, *WordsPerQuery* is very low. For example, systems that support only search by key or ID will have *WordsPerQuery* = 1. In these systems, *WordsPerQuery* - 1 is zero or very small, in which case HASH can have good performance because little data is transferred between servers.

Figure 20 shows overall performance in the same scenario as Figure 12, but with *WordsPerQuery* = 1.1. In this figure, we see that HASH is finally free of the inter-server bandwidth bottleneck for queries, and therefore has the best performance and scalability of all the strategies, particularly incremental HASH. Batch HASH is still bound by poor inter-server communication performance for logins, however, its performance is fairly scalable compared to the other strategies. In contrast, CHN starts off bound by user-server communication, but soon becomes bound by inter-server communication because of all the remote queries and remote results sent between servers. HASH does not suffer from this problem because, as discussed in Section 4, *ExServ* is fixed and *ExRemoteResults* is zero for HASH. In addition, FR is largely CPU bound, since login performance degrades rapidly as *NumServers* increases. As a result, HASH is clearly the recommended strategy when *WordsPerQuery* is close to 1.

Network Headers. In Section 6, we calculated the network bandwidth consumed by the system without considering network headers. The main reason for doing so was because we felt it was inefficient to send each AddFile message as a separate packet during login, and each QueryResponse message as a separate packet during query. Rather than redefine the network protocol, we decided to focus on the payload data, which varied between architecture and strategy, and defer the issue of headers until later. Now, we will discuss the impact of packet header size on system performance and show that while header size affects the absolute performance of the strategies, they do not affect their relative performance, which was the basis of our analysis on their strengths and weaknesses.

To simulate the effects of changing the network protocol, we can vary the average header size for the various message types. For example, a typical packet header is approximately 40 bytes for TCP/IP and Ethernet wrappers, and say we want to study the performance of the system if the protocol were defined such that an average of 10 AddFile messages were sent in a single packet. That is, instead of sending 168 AddFile packets to the server on login, the average client would only send 16-17 packets instead. To simulate the case where there are 10 messages per packet, we would evaluate system performance where average header size for an AddFile message is $40\frac{bytes}{packet}/10\frac{messages}{packet} = 4\frac{bytes}{message}$.

Figures 21 and 22 show the overall performance of each strategy when the header size for AddFile and QueryResponse is varied. We only analyze these two message types because for the music-sharing scenario, they account for over 98% of all user-server packets with the batch login policy, and over 95% with the incremental policy. All parameters are the defaults listed in Tables 1 and 2. In these figures, all packet header sizes other than the one being varied are assumed to be 40 bytes. As seen in these figures, packet header size does affect actual performance of the strategies, but does not affect the relative ordering of the strategies. Furthermore, the actual performance of most strategies is affected very little. Incremental and batch FR, which are bound by CPU, are not affected at all. The strategies that are most affected, batch and incremental CHN, are bound by user-server communication. User-server communication consists largely of many small packets, so a change in packet header size has a relatively significant impact on overall performance. However, note that HASH, which is bound by inter-server communication, is barely affected by a change in packet header size. This is because the bulk of the bandwidth used by HASH is consumed by payload data. Hence, packet header size has little or no affect on its overall performance.

In summary, we believe that network protocol and packet header issues are important to consider, to ensure the most efficient execution of the application. However, because our analysis in this section has been based on relative performance of the strategies and on characteristics of the performance graphs, we believe our analysis is unaffected by these issues.

8 Conclusion

In this paper, we studied the behavior and performance of hybrid P2P systems. We developed a probabilistic model to capture the query characteristics of these systems, and an analytical model to evaluate the performance of various architectures and policies. We validated both models using experimental data from actual hybrid P2P systems. Finally, we evaluated and compared the performance of each strategy. A summary of our findings for each architecture and policy are as follows:

- In our opinion, the **chained** architecture is the best strategy for today's music-sharing systems, and will continue to be unless network bandwidth increase significantly, or user interests become more diverse. This architecture has fast, scalable logins and requires the least amount of memory. However, query performance can be poor if many servers are involved in answering a single query.

- The **full replication** architecture has good potential in the future when network connections will be more stable and memory is cheaper. The architecture performs comparatively well in applications where user interests are diverse, and when result sets are relatively large.
- The **hash** architecture has very high bandwidth requirements. Hence, it is the best choice only if network bandwidth increases significantly in the future, or in systems where it is not necessary for servers to exchange large amounts of metadata (e.g., systems with search by key or ID, single-word queries).
- The **unchained** architecture is generally not recommended because it returns relatively few results per query and has only slightly better performance than other architectures. It is only appropriate when the number of results returned is not very important, or when no inter-server communication is available.
- The **incremental** policy is recommended in systems with negative correlation (e.g., historical or "archive-driven" systems), and performs best when sessions are short and network bandwidth is limited.

Acknowledgements We would like to thank Daniel Paepcke for administrating relations with the OpenNap network used for our studies.

References

[1] Freenet Home Page. http://freenet.sourceforge.com/.

[2] Gnutella Development Home Page. http://gnutella.wego.com/.

[3] ICQ Home Page. http://www.icq.com/.

[4] Konspire Home Page. http://konspire.sourceforge.com/.

[5] LOCKSS Home Page. http://lockss.stanford.edu/.

[6] Napster Home Page. http://www.napster.com/.

[7] OpenNap Home Page. http://opennap.sourceforge.net/.

[8] Pointera Home Page. http://www.pointera.com/.

[9] SETI@home Home Page. http://setiathome.ssl.berkely.edu/.

[10] Eytan Adar and Bernardo A. Huberman. Free Riding on Gnutella. http://www.firstmonday.dk/issues/-issue5_10/adar/index.html, September 2000.

[11] Eric W. Brown, James P. Callan, and W. Bruce Croft. Fast incremental indexing for full-text information retrieval. In *Proc. of 20th Intl. Conf. on Very Large Databases*, pages 192–202, September 1994.

[12] Brian Cooper, Arturo Crespo, and Hector Garcia-Molina. Implementing a reliable digital object archive. In *Proc. of the 4th European Conf. on Digital Libraries*, September 2000.

[13] Sandra Dykes. *Cooperative Web Caching: A Viability Study and Design Analysis.* PhD thesis, University of Texas at San Antonio, 2000.

[14] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kauffman Publishers, Inc., San Mateo, 1993.

[15] R. Guy, P. Reicher, D. Ratner, M. Gunter, W. Ma, and G. Popek. Rumor: Mobile data access through optimistic peer-to-peer replication. In *ER'98 Workshop on Mobile Data Access*, 1998.

[16] Brian Kantor and Phil Lapsley. Network News Transfer Protocol. RFC 977, February 1986.

[17] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. Web caching with consistent hashing. In *Proc. of the 8th Intl. WWW Conf.*, December 1999.

[18] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*, 14(4):349–379, October 1996.

[19] Anh NgocVo and Alistair Moffat. Compressed inverted files with reduced decoding overheads. In *Proc. of the 21st Intl. Conf. on Research and Development in Information Retrieval*, pages 290–297, August 1998.

[20] Michael Rabinovich, Jeff Chase, and Syan Gadde. Not all hits are created equal: Cooperative proxy caching over a wide area network. In *Proc. of the 3rd Intl. WWW Caching Workshop*, June 1998.

[21] Webnoize Research. *Napster University: From File Swapping to the Future of Entertainment Commerce.* Webnoize, Inc., Cambridge, MA, 2000. http://www.webnoize.com/.

[22] B. Ribeiro-Neto and R. Barbosa. Query performance for tightly coupled distributed digital libraries. In *Proc. of the 3rd ACM Conf. on Digital Libraries*, pages 182–190, June 1998.

[23] G. Salton. *Information Retrieval: Data Structures and Algorithms.* Addison-Wesley, Massachussetts, 1989.

[24] Anthony Tomasic. *Distributed Queries and Incremental Updates in Information Retrieval Systems.* PhD thesis, Princeton University, 1994.

[25] Anthony Tomasic, Hector Garcia-Molina, and Kurt Shoens. Incremental update of inverted list for text document retrieval. In *Proc. of the 1994 ACM SIGMOD Intl. Conf. on Management of Data*, pages 289–300, May 1994.

[26] J. Zobel, A. Moffat, and R. Sacks-Davis. An efficient indexing technique for full-text database systems. In *18th Intl. Conf. on Very Large Databases*, pages 352–362, August 1992.

A Probabilistic Query Model

A.1 Deriving $T(m, n)$: Probability of Retrieving m Results in a Collection of n files

For query q_i, the probability of retrieving m results in a collection of n files is the same probability of getting m successes in n Bernoulli trials, where the probability of success is $f(i)$. The expression for the probability is:

$$B(n,m) = \begin{pmatrix} n \\ m \end{pmatrix} (f(i))^m (1 - f(i))^{n-m}. \tag{7}$$

Because the probability of a success depends on which query is considered, we must take the expected value of this probability. Hence, $T(m,n)$, the expected value for $B(n,m)$, can be computed as:

$$T(n,m) = \sum_{i=0}^{\infty} g(i) \Big(\begin{pmatrix} n \\ m \end{pmatrix} (f(i))^m (1 - f(i))^{n-m} \Big). \tag{8}$$

A.2 Calculating ExServ for the Chained Architecture.

Let $P(s)$ represent the probability that exactly s servers, out of an infinite pool, are needed to return R or more results.

The expected number of servers is then:

$$ExServ = \sum_{s=1}^{k} s \cdot P(s) + \sum_{s=k+1}^{\infty} k \cdot P(s) \tag{9}$$

where k is the number of servers in the actual system. (The second summation represents the case where more than k servers from the infinite pool were needed to obtain R results. In that case, the maximum number of servers, k, will actually be used.)

Now, let us define $L(s)$ to be the probability that s *or fewer* servers are needed to return R or more results. Then, $P(s) = L(s) - L(s-1)$, and $L(0) = 0$ and $L(\infty) = 1$. The above expression for *ExServ* can be rewritten as:

$$ExServ = \sum_{s=1}^{k} s \cdot (L(s) - L(s-1)) + \sum_{s=k+1}^{\infty} k \cdot (L(s) - L(s-1)). \tag{10}$$

The first expression in Equation 10 can be expanded as follows:

$$\begin{aligned}
&\sum_{s=1}^{k} s \cdot (L(s) - L(s-1)) \\
&\quad = k \cdot L(k) - k \cdot L(k-1) + (k-1) \cdot L(k-1) - (k-1) \cdot L(k-2) ... 2 \cdot L(2) - 2 \cdot L(1) + L(1) - L(0) \\
&\quad = k \cdot L(k) - \sum_{s=1}^{k-1} L(s). \qquad (11)
\end{aligned}$$

Similarly, the second expression in Equation 10 can be expanded as follows:

$$\begin{aligned}
&\sum_{s=k+1}^{\infty} k \cdot (L(s) - L(s-1)) \\
&\quad = k \cdot (L(k+1) - L(k) + L(k+2) - L(k-1) + L(k+3) - L(k-2) ...) \\
&\quad = k \cdot L(\infty) - k \cdot L(k) \\
&\quad = k \cdot 1 - k \cdot L(k) \qquad (12)
\end{aligned}$$

Putting the two expressions together, we have:

$$\begin{aligned}
ExServ &= k \cdot L(k) - \sum_{s=1}^{k-1} L(s) + k - k \cdot L(k) \\
&= k - \sum_{s=1}^{k-1} L(s). \qquad (13)
\end{aligned}$$

Now, let $Q(n)$ be the probability that $R = \mathit{MaxResults}$ or more results will be returned from a collection of n or fewer files. Then, the probability that s or fewer servers are needed to return R or more results, $L(s)$, is equal to the probability that R or more results will be returned from a collection of at most as many files as are stored in s servers. That is, $L(s) = Q(s \cdot \mathit{UsersPerServer} \cdot \mathit{FilesPerUser})$. The probability $Q(n)$ of getting R or more results from a collection of n or fewer files is equal to the probability of *not* getting *fewer than* R hits from a collection of exactly n files. Hence, $Q(n)$ can be calculated as:

$$Q(n) = 1 - \sum_{m=0}^{R-1} T(n,m). \tag{14}$$

The final equation for ExServ for the chained architecture is therefore:

$$\begin{aligned} \mathit{ExServ} &= k - \sum_{s=1}^{k-1} \left(1 - \sum_{m=0}^{R-1} T(s \cdot \mathit{UsersPerServer} \cdot \mathit{FilesPerUser}, m)\right) \\ &= 1 + \sum_{s=1}^{k-1} \sum_{m=0}^{R-1} T(s \cdot \mathit{UsersPerServer} \cdot \mathit{FilesPerUser}, m). \end{aligned} \tag{15}$$

A.3 Calculating Expected Results for Chained Architecture.

For a given query q_i, the expected total number of results from a collection of n files is:

$$\sum_{m=0}^{R-1} B(n,m) \cdot m + \sum_{m=R}^{n} B(n,m) \cdot R \tag{16}$$

where $B(n,m)$ is defined in Appendix A.1 as the probability of getting m successes in n trials, where the probability of of a success is $f(i)$ – that is, the probability of getting m results in a collection of n files. Note that in the second term of the expression, if R or more results are found for a query, only R results are returned.

Equation 16 can be rewritten as:

$$\begin{aligned} \sum_{m=0}^{R-1} B(n,m) \cdot m + \sum_{m=0}^{n} B(n,m) \cdot R - \sum_{m=0}^{R-1} B(n,m) \cdot R \\ = \sum_{m=0}^{R-1} B(n,m) \cdot (m - R) + R \\ = R - \sum_{m=0}^{R-1} B(n,m) \cdot (R - m). \end{aligned} \tag{17}$$

Since the probability of a success in the above expression depends on which query is considered, we must take the expected value of this expression over all queries. Hence, $M(n)$, the expected value of the above expression, is computed as:

$$M(n) = \sum_{i=0}^{\infty} g(i)\Big(R - \sum_{m=0}^{R-1} B(n,m) \cdot (R-m)\Big). \tag{18}$$

A.4 Finding *ExServ* for the Hash Architecture

Let $P(b,i,k)$ be the probability that b inverted lists reside at exactly i servers when there are k total servers. Then, *ExServ* is equal to:

$$\mathit{ExServ} = \sum_{s=1}^{k} s \cdot P(\mathit{WordsPerQuery}, s, k) \tag{19}$$

Note that, unlike in the expression for *ExServ* in Appendix A.2 for the chained architecture, it does not make sense here to include expressions for $s > k$.

The problem of finding $P(b,i,k)$ can be restated as the probability of b balls being assigned to exactly i bins, where each ball is equally likely to be assigned to any of the k total bins. When b is small, we calculate this value directly. If b is a fraction, then we interpolate the value. For example, if $b = \mathit{WordsPerQuery} = 2.4$, then we first calculate *ExServ* for $b = 2$, then *ExServ* for $b = 3$. Let $k = 5$. Then, when $b = 2$,

$$P(2,1,5) = \binom{5}{1}\left(\frac{1}{5}\right)^2 = .2 \qquad P(2,2,5) = \binom{5}{2} 2! \left(\frac{1}{5}\right)^2 = .8. \tag{20}$$

Then, $\mathit{ExServ} = 1 \cdot .2 + 2 \cdot .8 = 1.8$.

Similarly, when $b = 3$,

$$P(3,1,5) = \binom{5}{1}\left(\frac{1}{5}\right)^3 = .04 \qquad P(3,3,5) = \binom{5}{3} 3! \left(\frac{1}{5}\right)^3 = .2 = 48 \tag{21}$$

and $P(3,2,5) = 1 - (P(2,1,5) + P(3.3.5) = 0.48$. Then, $\mathit{ExServ} = 1 \cdot .04 + 2 \cdot .48 + 3 \cdot .48 = 2.44$. Now, we interpolate a value for *ExServ* when $b = 2.4$:

$$\mathit{ExServ} = 1.8 + (2.44 - 1.8) \cdot (2.4 - 2) = 2.056. \tag{22}$$

When b is not small, then the expression for $P(b,i,k)$ becomes complex. In this case, we use a conservatively large approximation for *ExServ*. We know that if there is only one ball ($b = 1$), then $\mathit{ExServ} = 1$. If there are two balls ($b = 2$), then *ExServ* increases by a value of $1 - \frac{1}{k}$. As more balls are added, *ExServ* increases by an increasingly smaller value. However, we estimate *ExServ* by assuming for every ball added, *ExServ* will continue to increase by $1 - \frac{1}{k}$. For example, in the hash architecture, every word in a file's metadata is hashed to a server, and that file must be replicated at that server. To find at how many servers a file is expected to be replicated, we compute the value $1 + (\mathit{WordsPerFile} - 1)(1 - \frac{1}{k}) = 1 + (10 - 1)(1 - \frac{1}{5}) = 8.2$. We then take the maximum of this value, and the actual number of servers. Hence we expect a given file to be replicated at $max(8.2, 5) = 5$ servers on average. The estimate for *ExServ* will higher than the true value, but as long as k is small or b is not too large, the estimate will still be close.

B Analytical Performance Model

For the following example calculations, assume $UsersPerServer = 10000$, $r = 4$ and $\lambda_f = 100$.

B.1 Network Load

The messages in the Napster protocol that are part of our model as well are defined as follows:

1. `Login` message format: `(MsgSize MsgType <username> <password> <port> "<version>" <link-speed>)`.
2. `AddFile` message format: `(MsgSize MsgType "<filename>" <md5> <size> <bitrate> <frequency> <time>)`.
3. `RemoveFile` message format: `(MsgSize MsgType <filename>)`.
4. `Query` message format: `(MsgSize MsgType FILENAME CONTAINS "artist name" MAX_RESULTS <max> [FILENAME CONTAINS "file"] [LINESPEED <compare> <link-type>] [BITRATE <compare> "<br>"] [FREQ <compare> "<freq>"])`.
5. `QueryResponse` message format: `(MsgSize MsgType "<filename>" <md5> <size> <bitrate> <frequency> <length> <nick> <ip> <link-speed>)`.
6. `DownloadResponse` message format: `(MsgSize MsgType <nick> <ip> <port> "<filename>" <md5> <linespeed>)`.

If we estimate 10 characters in a user name or password, 7 characters to describe the file size, 4 characters to describe the bitrate, frequency, and time of an MP3 file, 1 character to describe a linespeed, 4 characters to describe a port, and 13 characters to describe an IP address, then the size of each message is:

1. `Login`: 42 bytes.
2. `AddFile`: 125 bytes.
3. `RemoveFile`: 67 bytes.
4. `Query`: 112 bytes.
5. `QueryResponse`: 140 bytes.
6. `DownloadResponse`: 131 bytes.

With these message sizes, we can now determine the network bandwidth consumed by login, query and download.

Login. In all architectures running the batch policy, when a user logs in, a Login message is sent to the server, as well as one AddFile message for every file in the user's library. Average bytes per login for user-server communication is:

```
LoginSize
= (Login Message Size) + FilesPerUser * (AddFile Message Size)
= 42 + 168 * 125
= 21042 bytes/login
```

In all architectures running the incremental policy, when a user logs in, a Login message is sent to the server, as well as one AddFile message for every file added to the user's library since the last logoff, and one RemoveFile message for every file removed from the user's library since the last logoff. From the parameters in Table 1, we can predict the number of files *changed* since the last login. We assume that half the changes are file deletions, and half the changes are file additions, since on average, this must be the case if the average number of files per user remains constant. Average bytes per login for user-server communication is:

```
LoginSize
= (Login Message Size) + FilesPerUser * FracChange * 0.5 *
(AddFile Message Size + RemoveFile Message Size)
= 42 + 168 * .1 * .05 * (125 + 67)
= 1654.8 bytes/login
```

For server-server communication, the bandwidth required for a login depends not only on policy, but on architecture as well. For the chained and unchained architectures, logins are not replicated across servers, hence 0 bytes/login are required. For the full replication architecture, login messages are sent to every server. If servers are connected via broadcast LAN, then the messages need only be replicated once, meaning 21042 bytes/login are needed for the batch policy, and 1654.8 bytes/login for the incremental policy. If servers are connected via point-to-point WAN, then all messages must be replicated $NumServers - 1 = 4$ times, meaning 84168 bytes/login are needed for the batch policy, and 6619.2 bytes/login for the incremental policy. Similarly, in the hash architecture, if servers are connected via LAN, then messages need only be replicated once, and the same bandwidth is required for server-server communication as for user-server communication. If servers are connected via WAN, then the login message must be replicated at every server in the chain, and the AddFile and RemoveFile messages must be replicated $s - 1$ times, where s, the number of servers a file hashes to, is calculated to be 5 by the method described in Appendix A.4. Average bytes per login is therefore 84168 bytes/login for batch, and 6619.2 bytes/login for incremental.

Query. With the given parameters, *ExServ* = 1.919, *ExRemoteResults* = 16.12 and *ExTotalResults* = 93.19. Then, the bytes per query required for user-server communication is, on average:

```
QueryLoad
= (Query Message Size) + ExTotalResults * (QueryResponse Message Size)
= 112 + 93.19 * 140
= 13158.6 bytes/query
```

This value holds for all architectures and both login policies.

For server-server communication, the chained architecture sends the Query message to as many servers as are necessary to satisfy the query – that is, $ExServ - 1 = 0.919$ servers. Also, a QueryResponse message is sent from remote server to local server for every remote result, and there are $ExRemoteResults = 16.12$ such remote results. Total bytes required for a query is therefore:

```
QueryLoad
= (ExServ-1) * (Query Message Size) + ExTotalResults * (QueryResponse Message Size)
= .919 * 112 + 16.12 * 140
= 2359.7 bytes/query
```

In the full replication and unchained architectures, servers do not send queries to each other, hence 0 bytes/query are required. In the hash architecture, the Query message must be sent out to the server containing the appropriate inverted lists. By the method described in Appendix A.4, we determine $ExServ = 2.12$. In addition, the inverted lists at the remote servers need to be transferred over to the local server.

Download. On download, a single Download message must be sent from user to server. The number of bytes per download, for all architectures and all login policies, is therefore:

```
DownloadLoad
= (Download Message Size)
= 131 bytes/download
```

In terms of server-server communication, in the chained and unchained architectures, servers do not notify each other of downloads; therefore 0 bytes/download are required. In the full replication architecture, every server must be notified of every download. Hence if servers are connected via LAN, 131 bytes/download are necessary. If servers are connected via WAN, $131 \cdot (NumServers - 1)$ $= 524$ bytes/download are necessary. Similarly, in the hash architecture, the file metadata must be replicated as many servers as it would during login. From our previous calculations using the given parameters, each file would be replicated at 5 servers. If servers are connected via LAN, 131 bytes/download are necessary. If servers are connected via WAN, $131 \cdot (5 - 1) = 524$ bytes/download are necessary.

B.2 CPU Load

Login. In the chained architecture, four main costs are incurred by a login: the startup cost/transaction overhead, the cost to read/process the incoming file metadata, the cost to write the file records to the database, and the cost to modify the inverted lists. Total cost is therefore $(a+b+c) \cdot \mathit{FilesPerUser} + d$, where a is the cost of reading/processing a file's metadata, b is the cost of writing a file record, c is the cost of modifying the inverted lists for one file's words, and d is the startup cost. Using rule-of-thumb values for in-memory transaction overhead and record read/write from [14], we estimate values for a, b and d to be 2000, 5000, and 100000 instructions respectively.

If n files' metadata are uploaded to the server on login, where $n = \mathit{FilesPerUser}$ if using the batch policy, and $n = \mathit{FilesPerUser} \cdot \mathit{FracChange}$ if using the incremental policy, then a total of $n \cdot \mathit{WordsPerFile}$ words are present in the uploaded metadata. We assume 60% of these words are distinct[8], and that all modifications that need to made to a list are done in one batch modification. The result of the overlap is that only $0.6 \cdot n \cdot \mathit{WordsPerFile}$ lists must be modified in total. By experimentally determining the cost of list writes, we determined that modifying a list requires approximately 11734.8 instructions per list modification. The value for coefficient c is therefore $0.6 \cdot \mathit{WordsPerFile} \cdot 11734.8 = 70408.8$. Note that this cost is independent of how long the list is.

Like login, logoff requires a startup cost. Furthermore, if the system implements the batch policy, every list that was modified during login must now be modified during logoff to remove the IDs of files belonging to the user, and all records of these files must be removed from the database. Total cost for logoff with the batch policy is therefore $(a + c) \cdot \mathit{FilesPerUser} + d$, where a, c and d have the same meanings as before. The cost for logoff with the incremental policy is simply d.

Putting the costs of login and logoff together, the total cost of login/logoff in the *batch* policy is:

```
LoginLogoffCost
= (a + 2b + 2c) * FilesPerUser + 2d
= 152817.6 * 168 + 200000
= 25873356.8 instructions/login-logoff
```

Total cost of a login/logoff in the *incremental* policy is:

```
LoginLogoffCost
= (a + b + c) * FilesPerUser * FracChange + 2d
= 77408.8 * 168 * 0.1 + 200000
= 1500467.84 instructions/login-logoff
```

Other architectures have similar formulas for login/logoff. In the unchained architecture, login/logoff follow the exact same procedure; hence cost is the same as in chained. In the full replication architecture, login/logoff also follow the exact same procedure, except that the procedure must be executed at every server. Hence, total instructions (across all servers) for login/logoff

[8]This is an approximation, since the amount of overlap in the collection of words would grow as the total grows.

is *NumServers* times the cost in the chained architecture: $5 \cdot 25873356.8 = 129366784$ instructions for the batch policy, and $5 \cdot 1500467.84 = 7502339.2$ instructions for the incremental policy. Finally, in the hash architecture, the login/logoff process is similar, except that every server might receive only a portion of the file metadata. With the given parameters, we have estimated in Appendix A.4 that every file is replicated at every server. Hence, the cost of a login/logoff is identical to that of full replication.

Query. Query has already been described in detail in Section 6.

Download. Download is similar to a login where the library consists of a single file. It involves the same equation as login: $(a + b + c) \cdot 1 + d$, where each coefficient has the same meaning. Each coefficient also has the same value, except for c. For login, we assumed that the words in the metadata had a significant overlap, so that only 60% of the total words were distinct. However, because there are so few words in a single file's metadata, we assume 0 overlap – that is, we assume every word is distinct. The value for c is therefore $\mathit{WordsPerFile} \cdot 11734.8 = 117348$. The total cost of a download is therefore:

```
DownloadCost
= (a + b + c) \cdot 1 + d
= (2000 + 5000 + 117348) \cdot 1 + 100000
= 222348 instructions/download.
```

Again, as for login, the cost of a download is multiplied by *NumServers* in the full replication architecture, since every server must process the download. Total cost is therefore $222348 \cdot \mathit{NumServers} = 1111740$ instructions/download. Similarly, in the hash architecture, the download must be processed at every server to which the metadata words hash. With the given parameters, the expected number of servers a file hashes to is *NumServers*, so the cost of a download in the hash architecture is $222348 \cdot \mathit{NumServers} = 1111740$ instructions/download.

C Memory Requirements

In the batch chained architecture, for every active user, the local server must keep a user record, estimated at 64 bytes. For every file in the user's library, the server must keep a file record, estimated at 128 bytes. Furthermore, every file has $\mathit{WordsPerFile} = 10$ words in the metadata, and for every word, an ID for that file is stored in the appropriate inverted list, where an ID is estimated at 8 bytes. Hence, total memory usage per active user is:

```
MemoryUsage
= 64 + 128 * FilesPerUser + 8 * WordsPerFile * FilesPerUser
= 64 + 128 * 168 + 8 * 10 * 168
= 35008 bytes/user
```

The unchained architecture implementing the batch policy requires the same amount of memory as the chained. In the full replication architecture, every user's information is replicated at each server, so that total memory usage is $\mathit{NumServers} \cdot 35008 = 175040$ bytes per active user. In the hash architecture, every user record is replicated at each server. In addition, given the current parameter values, every song is also replicated at each server (but this would not be the case if *NumServers* were larger than *WordsPerFile*). However, inverted lists are *not* replicated across servers, so for every word in the file metadata, only one ID for that file is stored in the appropriate inverted list, as in chained. Total memory usage per active user in hash is therefore:

```
MemoryUsage
= 64 * NumServers + 128 * FilesPerUser * NumServers + 8 * WordsPerFile * FilesPerUser
= 64 * 5 + 128 * 168 * 5 + 8 * 10 * 168
= 121280 bytes/user
```

For all architectures, if the incremental policy is implemented rather than the batch policy, then every server holds $\frac{1}{\mathit{ActiveFrac}}$ times as many users as are held with the batch policy. Hence, memory usage is $\frac{1}{\mathit{ActiveFrac}}$ times greater per active user.