Paper Number 473

Comparing Hybrid Peer-to-Peer Systems

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 [7], Gnutella [3] and Freenet [2], users directly exchange music files. In instant messaging systems like ICQ [4], users exchange personal messages. In systems like Seti-at-home [9], computers exchange available computing cycles. In preservation systems like LOCKSS [6], 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 Scour, 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 they be "chained," so that if one fails to get sufficient answers to a query, it can contact other servers? Should indexes be replicated at multiple servers? What types of queries do users typically submit in such systems? How many files do users typically share? How should the system deal with users that are disconnected often (dial-in phone lines)? What are the main system parameters that affect scalability and performance?

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 [8]. 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) a quantitative comparison of file-sharing hybrid P2P architectures, based on our query and performance models.

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 and discuss in further detail the steps we took to validate our models.

Also note that by studying hybrid P2P systems, we do not take any position regarding their *legality*. Some P2P systems like Napster and Scour 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 system 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 [2] describes the Freenet project which was designed to provide anonymity to users and to adapt to user behavior. References [12] and [13] describe systems that use 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 (e.g., [18, 24]) and overall IR system performance (e.g., [17, 26]). However, while the IR domain has many ideas applicable to P2P systems, optimization techniques cannot be directly applied because of the large differences in update frequency between the two systems. 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]), the techniques do not scale to the rate of change experienced by P2P systems.

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 [8] server implementation, and resembles the "local inverted list" strategy [21] 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 [5]. The "hash" architecture resembles the "global inverted list" strategy [21] 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 [7].

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

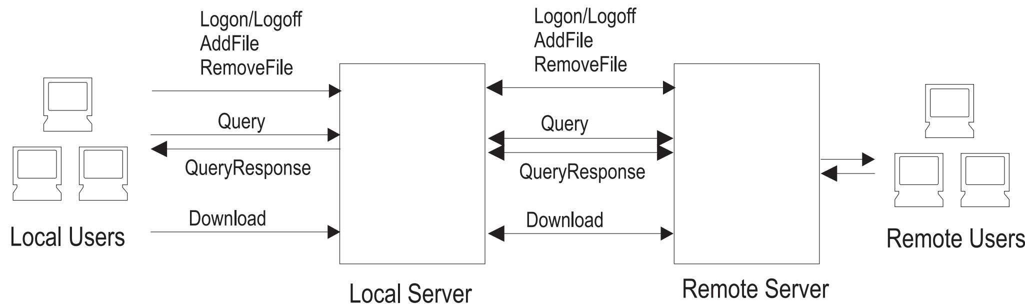

Figure 1: System Components and Messages

3 Server Architecture

We begin by describing the simple "chained" architecture, based on the OpenNap [8] implementation of a file-sharing service.[2] All other architectures will be variations of this one. Figure 1 shows the components in the basic hybrid P2P system and how they interact.

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. 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 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*.

The servers form a linear chain that is used in answering queries. When a user submits a query to her local server, the server searches for matches in its index. The server sets a maximum number of results to return for any query. If the server cannot find the maximum number of results, it will forward the query to a remote server along the chain, in hopes that the next server will find more results. The remote server will return any results back to the first server, which will then forward the results to the user. The server continues to send the query out to the remaining servers in the chain until the maximum number of results have been found, or until all servers have received and serviced the query. A query is said to be *satisfied* if the maximum number of results are returned.

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 server is also notified when local files are deleted.

[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.

[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	400	Inverse frequency skew (to be described in Section 4)
r	10	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

Upon logoff, the local server updates the index to indicate that the user's files are no longer available. The options for handling logoffs are discussed in the next subsection.

The index maintained by the server takes the form of inverted lists [22]. Every file's metadata is considered a document, with the text of the file name, author name, and so on, being its content. A query consists of a list of desired words. 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 querying schemes are possible (e.g., relational queries), but since they are not used in today's P2P systems, we do not consider them here.

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 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.

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

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 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.

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 [23]; 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).

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). Thus, we assume 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 popular files are *queried* for frequently, and *stored* frequently. 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 our model, 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 [20] 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, where the selection power of q_i may depend on past popularity of q_i.

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

Calculating ExServ for the Chained Architecture. Using our model we now compute *ExServ*, the expected number of servers needed in a chained architecture (Section 3) to obtain the desired $R = MaxResults$ 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 [1] we show that this expression can be rewritten as:

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

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:

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

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

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

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 [1] 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 [1] 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* · *FilesPerUser* is the number of files at a *single* server.

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 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.

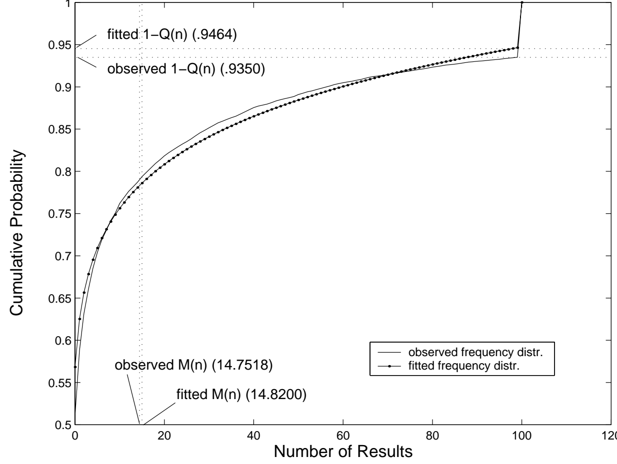

Figure 2: Observed and Fitted Frequency Distributions of Results in OpenNap

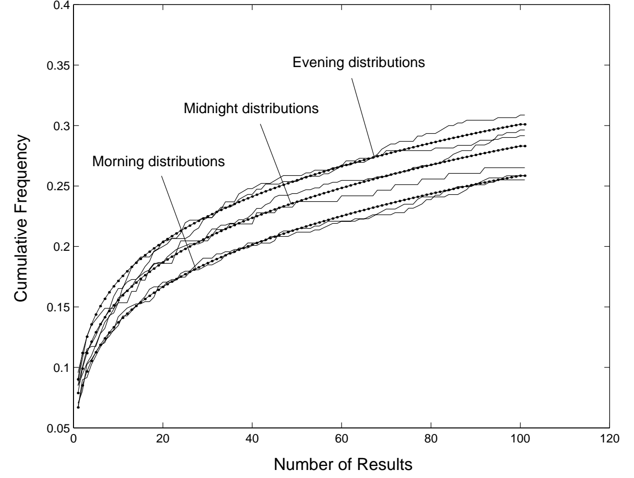

Figure 3: Observed and Fitted Frequency Distributions of Results in Napster

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 2 shows the cumulative normalized version of that 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 2 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 2 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 2 should match the function $h(x) = \sum_{m=1}^{x} T(69000, m)$ in the range $1 \leq x \leq 99$. 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 2 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 Figure 2). 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

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 4: Characteristics of Napster Query Runs

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 4 shows the results for these experiments, together with the λ_f and r values obtained by 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 3, where the observed Napster distributions are marked in solid lines, and the fitted distributions in dotted lines.[4]

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.

[4] 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.

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.

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 [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 logs on approximately 8 times a day. This number may seem high at first, but given the extremely low *QueryLoginRatio* (see discussion above), the number of logins a day per user should be proportionally high. 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 [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

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 5: Formulas for cost of actions in CPU instructions

refer to [1].

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. We assume the LAN is a broadcast network whereas WAN connections are point-to-point. 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 5 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 tests using simple emulation code. Finally, we experimentally validated the overall formulas against OpenNap performance.

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

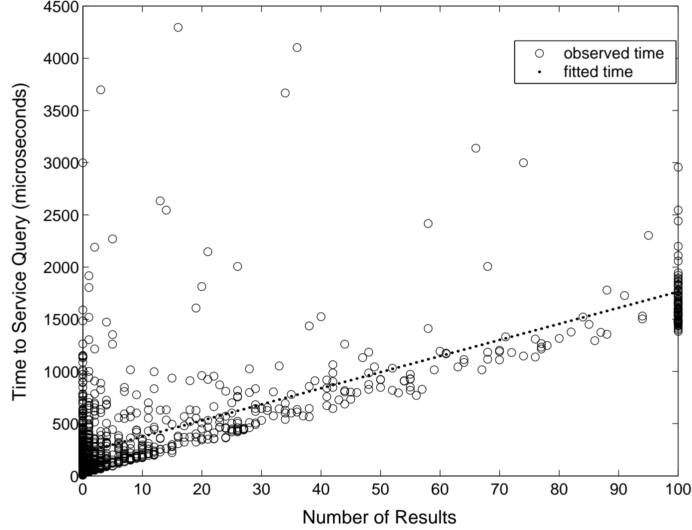

Figure 4: Time to Service a Query vs. Number of Results Returned

cost of reading the local results, and the cost of transferring remote results from 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 $a \cdot ExTotalResults + b \cdot ExRemoteResults + c \cdot ExServ$, where c is the startup cost, a is the cost associated with finding one result, and b 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 5 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 4 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 $ExServ = 1$, and $ExRemoteResults = 0$. Figure 4 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 4) 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 result needs to be divided by $ActiveFrac$.

Action	Formula
Batch Login	$42 + (75 + \mathit{WordsPerFile} \cdot \mathit{CharPerWord}) \cdot \mathit{FilesPerUser}$
Incr. Login	$42 + (92 + 2 * \mathit{WordsPerFile} \cdot \mathit{CharPerWord}) \cdot \mathit{FilesPerUser} \cdot \mathit{FracChange}$
Query	$\mathit{WordsPerQuery} \cdot \mathit{CharPerWord} + 100$
QueryResponse	$90 + \mathit{WordsPerQuery} \cdot \mathit{CharPerWord}$
Download	$23 + \mathit{WordsPerFile} \cdot \mathit{CharPerWord}$

Table 6: Formulas for cost of actions in bytes

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 6 lists formulas for the cost, in bytes, of the messages required for every action in the simple architecture. The coefficients come directly from the Napster network protocol, user-defined parameters, and a number of small estimates.

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.[5] 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 $42 + 168 \cdot 0.2 \cdot 0.5(125 + 67) = 3267.6$ bytes.

Network usage between servers varies depending on the architecture. For login, the unchained architecture has no inter-server communication, so the required bandwidth is 0. Servers in the chained architecture send queries between servers, but not logins, so again, the bandwidth is 0. With the full replication architecture,

[5] 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.

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 [1] 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 resources, 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[6] 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 6, we know the cost of a login is 21546 bytes, the cost of a query is 112 bytes, and the cost of a download is 125 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 340.2 bits per user per second, meaning unused bandwidth is 7.6598 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 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

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

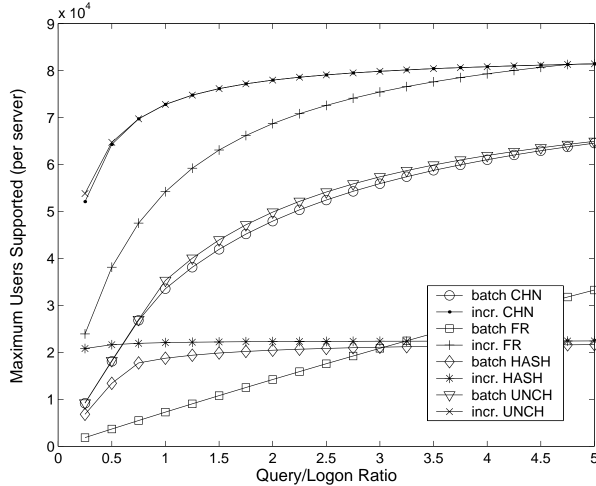

Figure 5: Overall Performance of Strategies vs. *QueryLoginRatio*

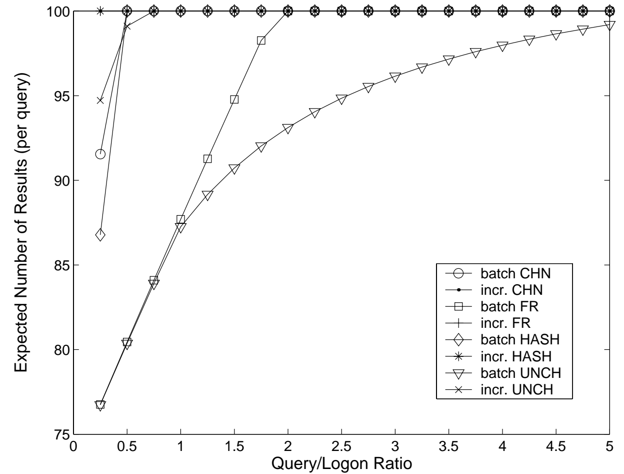

Figure 6: Expected Number of Results

in different scenarios. Throughout the performance comparisons, the metric of performance is the maximum number of users each server can supported. 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 implementing 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 Modeling Napster

We begin by evaluating performance of systems with query behavior similar to that of Napster, described by query model parameters $r = 4$ and $\lambda_f = 100$ (see Section 5). Figure 5 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.

As shown in this figure, 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 5 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 6 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. In summary, 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.

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

Batch versus Incremental CPU Performance. In Figure 5, 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 5, 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 7, 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, when *QueryLoginRatio* is small, the incremental strategies perform better than batch because the balance of queries to logins is in their favor. However, as *QueryLoginRatio* increases, the balance of favor shifts, and batch strategies continue to improve until they surpass the incremental strategies in performance.

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 7.2.

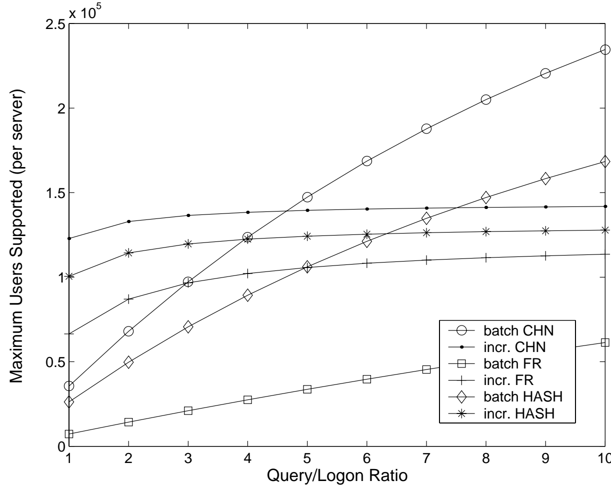

Figure 7: CPU Performance of Login Policies vs. *QueryLoginRatio*

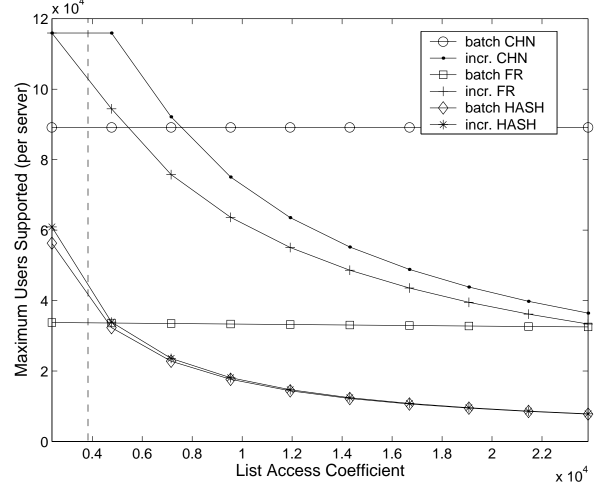

Figure 8: Overall Performance of Strategies vs. List Access Coefficient

7.2 Modeling OpenNap

Thus far, we have been evaluating strategies in Napster's query model ($r = 4, \lambda_f = 100$). Now, we will consider OpenNap's query model ($r = 10, \lambda_f = 400$). Unless otherwise specified, we will set the default value for *QueryLoginRatio* to be 5, a value that more clearly highlights differences between strategies. We repeated the experiments in the previous section on the effects of *QueryLoginRatio* on performance, and observed the same trends. Therefore we will now focus on the impact of other parameters on system performance.

Performance Impact of List-access Coefficient. Here, we will study the effects of the list-access coefficient used in the formulas for query costs in Table 5. This coefficient describes the number of instructions per result required to access the inverted lists. We believe this coefficient has the highest likelihood to vary between systems because of the implied ratio of the number of list accesses and the number of results returned. In the music industry, many queries contain proper nouns (e.g., artist's names) and other "focused" words that would result in relatively few list accesses for every result. On the other hand, queries in other types of systems, such as a news article database, would often consist of very general, common words with a relatively small intersection (e.g., "president campaign Florida"). As a result, we expect many more list accesses to scan through the long inverted lists for a relatively small number of results, and the coefficient would be higher.

Figure 8 shows the effect of the list-access coefficient on overall performance. The coefficient derived in Table 5 for the OpenNap system is 3778, marked by a dotted line in the figure. A higher value for the coefficient, for example, 7556, means that twice as many instructions are required per result to access the inverted lists.

Incremental strategies are most negatively affected by increases in the coefficient, since the negative effect on query performance is magnified by $1/ActiveFrac$ (see Table 5). Batch CHN and incremental CHN

both start off bound by user-server communication, but as the value for the coefficient increases beyond 2, 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.

Performance Impact of Query Model Parameters. In Section 4, we developed a model for user query behavior, parameterized by λ_f and r. Here, we discuss the effects of these parameters on performance. Due to lack of space, we consider only r, which as a larger impact on performance than λ_f.

Recall that r is the ratio between the inverse skew factors of query frequency and query selectivity power. As r increases, the relative skew of query frequency to selectivity power decreases. As a result, *ExTotalResults* decreases, and *ExServ* increases. We hypothesized the Napster system ($r = 4$) has a smaller r value than the OpenNap system ($r = 10$) because the Napster community tended to query for more trendy files. We can imagine other applications where r can be much higher, or lower, then that observed in OpenNap. For example, a system in the business domain where analysts share market research may have evenly distributed queries across all files, because the firm has clients from all different sectors of the market. With such evenly distributed queries, the r parameter for this system would be quite high.

Figure 9 shows overall performance as r is varied. The r value derived for the OpenNap system is shown by the dotted line 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.

For the same reason, we see that CHN and HASH show greater improvement than FR. FR, which has poor login performance but excellent query performance, is least affected by a decrease in *ExTotalResults* caused by an increase in r.

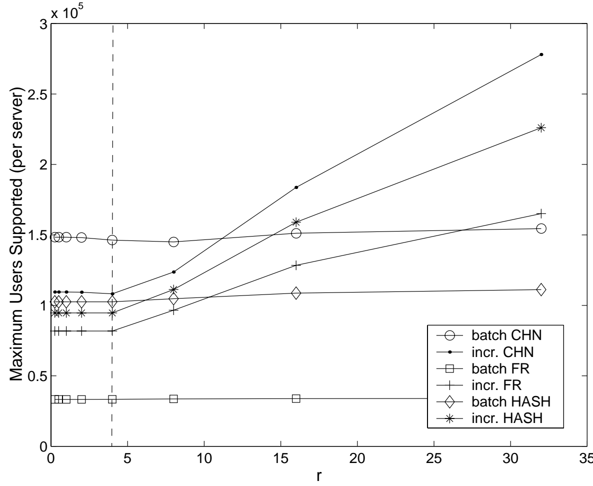

Figure 9: Overall Performance of Strategies vs. Query Model Parameter r

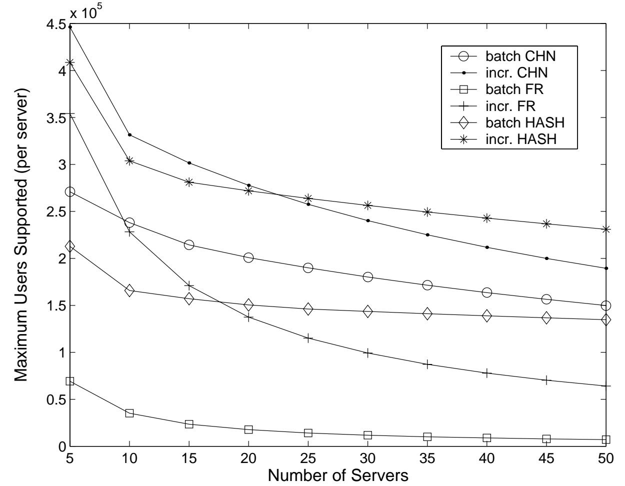

Figure 10: CPU Performance of Strategies vs. Number of Servers in System

Scaling Number of Servers. For this study, we will consider a likely future scenario where more users have stable, high-speed network connections and can stay online for longer periods of time. The result is a higher *QueryLoginRatio* and *ActiveFrac*. Also, we assume in this scenario that query frequency is distributed, and users typically submit several queries before locating the desired materials to download. Such characteristics are likely in a system such as the business research-sharing application mentioned earlier, where queries are very distributed and it may take several attempts at a query before sufficiently narrowing down the search. The result is a higher *QueryDownloadRatio* and r.

Figure 10 shows the CPU performance of architectures in the scenario we described, as the number of servers in the system is scaled up. Here, *QueryLoginRatio* = 10, *ActiveFrac* = .25, *QueryDownloadRatio* = 2, and r = 16. 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.

As seen in this figure, HASH is 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 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 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.

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. As number of servers increases, logins take a proportionally large

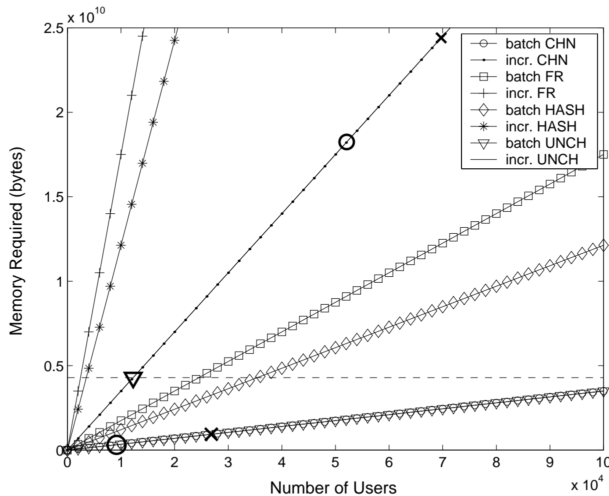

Figure 11: Memory Requirements

slice of the CPU.

7.3 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 11 shows the memory requirement in bytes of the various strategies, as a function of the number of users. 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 11. Now, consider a Napster scenario where $r = 4$, $\lambda_f = 100$, and $ActiveFrac = .1$. Say we determine that $QueryLoginRatio = .75$. 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 11 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 11, 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 11 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.

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 (including results in [1]) for each architecture and policy are as follows:

- The **chained** 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. This architecture is good when sessions are short (such that *QueryLoginRatio* is low), and the popularity and selectivity skew of queries is high (such that *ExServ* is low).

- The **full replication** architecture has excellent scalable query performance, but slow, unscalable logins and a high memory requirement. This architecture performs best when sessions are long or when network bandwidth is limited.

- The **hash** architecture has scalable queries and logins, but very high bandwidth requirements. Hence, the architecture performs well when there are many servers in the system, and few words per query.

- The **unchained** architecture 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 has fast logins but slow queries. Hence, it performs well when sessions are short (such that *QueryLoginRatio* is low). This policy requires more memory than the batch policy.

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

References

[1] Extended technical report. Reference removed for double-blind reviewing.

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

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

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

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

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

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

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

[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] Yuan Chen, Jan Edler, Andrew Goldberg, Allan Gottlieb, Sumeet Sobti, and Peter Yianilos. A prototype implementation of archival intermemory. In *Proc. of the 4th ACM Conf. on Digitial Libraries*, 1999.

[13] 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.

[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] Patrick Martin, Ian A. Macleod, and Brent Nordin. A design of a distributed full text retrieval system. In *Proc. of the ACM Conf. on Research and Development in Information Retrieval*, pages 131–137, September 1986.

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

[21] 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.

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

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

[24] Anthony Tomasic and Hector Garcia-Molina. Performance of inverted indices in shared-nothing distributed text document information retrieval systems. In *Proc. of the 2nd Intl. Conf. on Parallel and Distributed Information Systems*, pages 8–17, January 1993.

[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.