Adaptive Content Management in Structured P2P Communities

Jussi Kangasharju
Dept. of Computer Science
TU Darmstadt
Darmstadt, Germany
jussi@tk.informatik.tu-darmstadt.de

Keith W. Ross
Dept. of Computer and Information Science
Polytechnic University
Brooklyn, NY
ross@poly.edu

David A. Turner
Dept. of Computer Science
CSU San Bernadino
San Bernardino, CA
dturner@csusb.edu

ABSTRACT

A fundamental paradigm in P2P is that of a large community of intermittently-connected nodes that cooperate to share files. Because nodes are intermittently connected, the P2P community must replicate and replace files as a function of their popularity to achieve satisfactory performance. We develop a suite of distributed, adaptive algorithms for replicating and replacing content in a P2P community. We do this for structured P2P communities, in which a distributed hash table (DHT) substrate is available for locating the node responsible for a key. In particular, we develop the Top-K MFR replication and replacement algorithm, which is not only straightforward to layer on top of a DHT substrate, but also adaptively converges to a nearly-optimal replication profile. Furthermore, we develop an analytical optimization theory for benchmarking the performance of replication/replacement algorithms, including algorithms that employ erasure codes.

1. INTRODUCTION

One of the most compelling uses of the Internet today is P2P file sharing of multimedia content. Popular P2P file sharing systems, such as KaZaA, support millions of simultaneous users, and provide sharing of a variety of file types, including large multimedia files such as MP3s (typically in 3-6 Mbyte range) and videos (ranging from 5 Mbytes to multiple Gigabytes) [1, 2]. Today, P2P file sharing is the dominant traffic type in the Internet, exceeding that of all other applications, including the Web.

The file sharing systems KaZaA and Gnutella are often referred to as "unstructured" P2P systems because (i) the nodes are not organized into highly-structured overlays, and (ii) content is (essentially) randomly assigned to nodes. Because content is randomly assigned to nodes, the unstructured P2P systems must resort to limited-scope search for locating content. Unfortunately, a limited-scope query may not find a node with the desired file even if the file is present in some node in the P2P system. Furthermore, the unstructured file sharing systems do not attempt to replicate/replace content in a manner that is socially advantageous for the P2P community at large.

Recently, a number of proposals have been put forth for "structured" P2P systems, including CAN [11], Chord [12], Pastry [9], and Tapestry [16]. Structured P2P systems use *distributed hash table (DHT) substrates*, which organize nodes into highly-structured overlay networks and which deterministically assign keys to nodes. A DHT substrate can serve as a platform for a variety of P2P applications, including persistent file storage [3] [4] [5], multicast [6] [7], mobility management [6], and Web caching [8]. DHT substrates are also compelling platforms for P2P file sharing of multimedia content, since they can provide efficient file location procedures.

In this paper we examine DHT-based *file-sharing communities*. A P2P file-sharing community is a collection of intermittently-connected nodes with each node contributing storage, content and bandwidth to the rest of the community. When a node in the community wants a particular file, it first attempts to retrieve the file from the other nodes in the community. If the desired file is not found in the community, the community retrieves the file from the outside, possibly caches the file, and forwards a copy to the requesting node.

As an example of a P2P file-sharing community, consider a university campus network. As shown in Figure 1(a), the peers in a campus are typically interconnected with a high-speed LAN, and the high-speed LAN is connected to the global Internet via a lower-speed access link. Currently, the peers within a campus do not organize themselves as a P2P community: the peers in the campus independently retrieve the same popular music and video content from peers outside the campus, clogging the access links and wasting peer storage. The university campus could make more efficient use of its resources (WAN bandwidth and peer storage) if the peers were organized in a P2P community. As a P2P community, the peers in the campus would collectively maintain a managed number of copies of files, and would attempt to retrieve files internally before retrieving them from outside

the campus.

A second example of a P2P file-sharing community is a *content-distribution booster*, for example, for the distribution of video training content in a large corporation. As shown in Figure 1(b), videos are permanently archived in a small number of servers, which collectively do not have enough aggregate server and/or transmission capacity to serve all the users in the corporation. By organizing all the corporate nodes into a P2P community, the P2P community serves as a front-end surrogate for video distribution. Popular videos would often be downloaded (or streamed) from other corporate peers, thereby relieving the burden on the archival servers.

In this paper we address the problem of content management in P2P file-sharing communities. Our focus is in on the sharing of large audio and video files. We study the problem in a DHT context, that is, the individual nodes in the community are coupled together with a DHT substrate. Throughout we make the natural assumption that intra-community file transfers occur at relatively fast rates as compared with file transfers into the community; this is clearly the case for campus networks and content-distribution boosters.

The essence of our problem is to adaptively manage content in a P2P community to minimize the average delay, which is defined as the time from when a node makes a query for a file until the node receives the file in its entirety. Because the file transfer delay is typically orders of magnitude larger than lookup delays, and because intra-community file transfers occur at relatively fast rates, our problem is, for all practical purposes, equivalent to adaptively managing content to maximize intra-community hit rates. (We discuss this equivalence in more detail in Section 3.)

There are two important issues in maximizing the intra-community hit rate.

- **Replication:** Because nodes connect and disconnect to the network (or to the "application"), to provide satisfactory hit rates, content needs to be replicated across multiple nodes in the community. Naturally, popular content needs to more aggressively replicated than unpopular content. At the same time, content should not be excessively replicated, wasting bandwidth and storage resources.[1]

- **File Replacement Policies:** Each participating node has a limited amount of storage that it can offer to the community. Even with, say, 100 Gbytes of shared storage per node, a node will not be able to store more than 20-40 DVD videos. When this storage fills at some node, the node needs to determine which files it should keep and which it should evict.

The principal contribution of this paper is a series of algorithms for dynamically replicating and replacing files in a P2P community. These algorithms make no *a priori* assumptions about file request probabilities or about nodal up probabilities. They are therefore appropriate for when file request probabilities are changing over time and new files are being introduced in the system daily. The algorithms are simple, adaptive and fully distributed. They can ride on top of any of the DHT substrates (e.g., [11, 12, 9, 16]).

We first propose natural and intuitive Least Recently Used (LRU) Algorithms. Although the LRU algorithms provide better performance than non-cooperative schemes, their performance remains well below that of a provable upper bound. We then devise an alternative algorithm, called *Top-K Most Frequently Requested* (Top-K MFR), which through simulation analysis is shown to give remarkably good performance - nearly optimal for all tested scenarios. We also provide an efficient and accurate procedure for analytically evaluating the steady-state performance of the Top-K MFR algorithm. This analytical procedure further confirms the near optimality of the Top-K MFR algorithm.

A second important contribution of this paper is an analytical optimization theory for replication in P2P communities. For the analytical theory, we assume that file popularities and nodal up probabilities are known *a priori*. We develop the theory for complete-file replication as well as for the case when files are segmented and erasure codes are used to provide redundancy. For the general case with erasures, we show that a simple, separable concave optimization problem provides an upper bound on the performance of all adaptive schemes. For complete file replication, we show that an explicit *logarithmic assignment rule* is optimal. The optimization theory provides significant insight into managing P2P content, and also allows us to benchmark the adaptive replication algorithms.

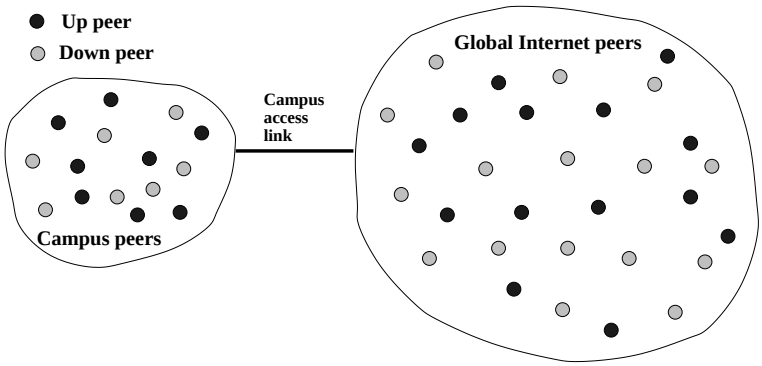
This paper is organized as follows. Section 2 reviews related work. In Section 3 we propose and analyze distributed content management algorithms, including Top-K LRU and Top-K MFR. In Section 4, we develop the analytical theory for optimal replication in P2P, including a theory for replication with erasures. In Section 5 we conclude and discuss future directions.

2. RELATED WORK

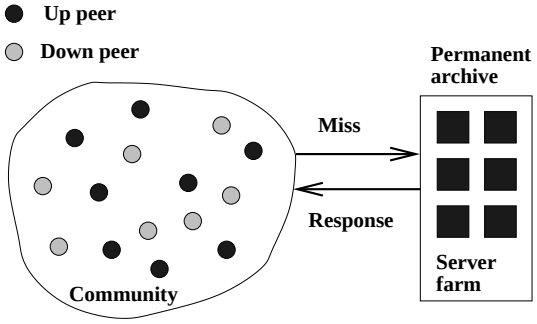
A P2P community can also be viewed as a distributed P2P cache for caching large multimedia files. Squirrel [8] is a recent proposal and implementation of a distributed, serverless, P2P Web caching system. Squirrel, which is built on top of the Pastry [9] DHT substrate, has been carefully designed to serve as an alternative for a traditional Web proxy cache. While such detailed protocol design and implementation issues are clearly important, the work [8] has not focused on the fundamental issues of replication and file replacement in a P2P community. Furthermore, the focus of [8] is on Web objects, whereas the focus of this paper is on large multimedia files, which account for the majority of the traffic in today's file sharing systems.

FarSite [21] (see also [22] [23]) is a P2P file system with the strong persistence and availability of a traditional file system. The FarSite filesystem uses the same number of replicas – three – for each file. In contrast with a file system, the

[1]It is problematic when thousands of students on the same campus download and store the same recently-released movie [15].



(a) Campus community



(b) Content distribution booster

Figure 1: Different types of P2P communities

goal of a P2P community is not to provide strong file persistence, but instead maximal content availability. Thus, in a P2P community, the number of replicas of an file depends on the popularity of the file.

Lv et al [13] and Cohen and Shenker [14] studied optimal replication in an unstructured peer-to-peer network in order to reduce random search times. Our work differs in that we are replicating content in structured, DHT-based networks, and we take intermittent connectivity explicitly into account. Furthermore, we replicate to decrease average file transfer time rather than to decrease the random search time in an unstructured P2P system.

There has also been work comparing replication and erasure coding in distributed storage infrastructures [10]. The authors also discuss the drawbacks of using erasures. The paper focuses on persistent storage, and does not consider adaptive replication and replacement of replicas or erasures.

3. ADAPTIVE ALGORITHMS FOR CONTENT MANAGEMENT IN P2P COMMUNITIES

As discussed in the Introduction, a P2P community consists of a large community of intermittently-connected nodes that cooperate to share content. The nodes in the community could be workstations, desktop PCs, portable PCs, etc.[2]

[2]Low-bandwidth, low-storage devices would not likely cache and serve files in the community (although they may be permitted to download files from the community).

Each participating node allocates a fraction of its storage to the P2P community. We refer to this allocation as the node's *shared storage*. (The nodes also have private storage, which is not accessible by the other nodes in the community.) We suppose that the files in a node's shared storage are not lost when a node disconnects; when a peer comes back up, its files again become available. (This is generally the case in P2P file-swapping systems such as KaZaA and Gnutella.)

As mentioned in the Introduction, a natural performance measure is *average delay*, where delay is defined from the moment a request is made until the entire file is downloaded. Delay has two components: the time to locate a copy of the file, and the time to download the file. Because our focus is on large files (e.g., music and video), the delay is dominated by the downloading component. Because intra-community file transfers occur at relatively fast rates, the downloading delay is directly correlated to the probability of finding the file within the community, that is, the intra-community hit probability. Henceforth, our optimization criterion is to maximize the hit probability[3].

3.1 DHT Substrate

Each node has a persistent identifier, which is assigned when the node initially subscribes to the application. (Because nodes typically change their IP addresses each time they come up, the IP address cannot be used as the persistent

[3]In some circumstances, it can be argued that the byte hit probability is more appropriate than the file hit probability. The algorithms and theory developed in this paper are easily modified for byte hit performance.

identifier.)

As shown in Figure 2, our algorithms require that each participating node has access to the API of a DHT substrate. The substrate has a function call that takes as input a file identifier j and determines an ordered list of the up nodes. The substrate then returns, for a given value of K, the first K nodes on the list, $i_1, i_2, \ldots, i_K$. The node i_1 is said to be the current first place winner for file j; the node i_2 is said to be the current second-place winner for file j, and so on. There are a number of substrates today that provide this functionality for first-place winners, including CAN [11] Chord [12], Pastry [9] and Tapestry [16]. These substrates are easily extended to provide the top K winners.

3.2 Top-K LRU Algorithm

We now begin to address the following fundamental problem in structured P2P systems: How can we adaptively add and remove replicas, in a distributed manner and as a function of evolving demand, to maximize the hit probability?

Our approach is to couple the file location provided by the DHT substrate with on-the-fly replication and replacement. The basic idea is as follows. When there is a request for a file j, the community uses the DHT substrate to search for a replica in the community; if the community does not find a replica in an up node, a new replica of j is obtained and stored in the current first-place node for j and a replica of another file is (possibly) evicted from the node. This simple idea is at the core of our adaptive algorithms. However, we shall see that a naive application of the idea gives unsatisfactory performance, but that a collection of subtle, yet critical, refinements provide near-optimal performance.

We begin with a simple, intuitive adaptive algorithm. Suppose X is a node that wants file j. X will obtain j as follows:

Basic LRU Algorithm

1. X uses the substrate to determine i_1, the current first-place winner for j.
2. X asks i_1 for j. If i_1 doesn't have j (a "miss" event), i_1 retrieves j from outside the community and puts a copy in its shared storage. If i_1 needs to evict a file to make room for j, i_1 uses the LRU replacement policy.
3. i_1 sends j to X (either for streaming or for downloading into X's private storage). Note that X does not put j in its shared storage unless $X = i_1$.

One obvious problem of the Basic LRU Algorithm is that a request can be a "miss" even when the file is cached in some up node in the community. Indeed, suppose file j is cached at the first-place winner and then, just prior to a request for file j, a new node comes up which becomes the new first-place winner for j. Then the Basic LRU Algorithm will retrieve file j from the outside even though it is cached in the community. To mitigate this problem, we modify the Basic LRU Algorithm as follows. In Step 2, when i_1 doesn't have j, i_1 determines $i_2, \ldots, i_K$ and pings each of these $K-1$ nodes to see if any of them have j. If so, i_1 retrieves j from of them and puts a copy in its shared storage. Otherwise, as before, i_1 retrieves j from outside the community. We refer to this modified algorithm as the **Top-K LRU Algorithm**.

Observe that the Top-K LRU Algorithm replicates content without any *a priori* knowledge of file request patterns or nodal up probabilities. It is also fully distributed. Although it is still possible that there will be a miss when the desired file is in some up node in the community, we will show that if K is appropriately chosen, the probability of such a miss is negligibly small.

To determine the hit performance of our adaptive algorithms, we have run simulation experiments with 100 nodes and 10,000 files. Studies in caching and P2P have consistently confirmed that request probabilities follow a Zipf distribution [17, 18]. Our simulations also use a Zipf distribution with parameters .8 and 1.2 [18].

In the simulation experiments reported here, all file sizes are of the same size. (We also did extensive experiments with heterogeneous file sizes and obtained similar results.) Because all files are of the same size, the byte hit probability is equal to the hit probability. In the simulation experiments reported here, each node contributes the same amount of shared storage to the community. (We also did extensive experiments with heterogeneous storage, and obtained similar results.) Our experiments run from 5 files per node to 30 files per node.

For the case of homogeneous up probabilities, we will derive the theoretical optimal policy and corresponding upper bound (over the set of all replication/replacement policies) in Section 4. Because such a bound is available, we can use it to benchmark adaptive algorithms when all nodes have the same up probability. All of the experimental results we report here use homogeneous up probabilities for the nodes. We have considered two up probabilities: .2 and .9. We have also performed testing with heterogeneous up probabilities (that is, different nodes having different up probabilities), and have found that our algorithms have similar performance behavior.

Figure 3 shows four graphs, one for each of the combinations of Zipf parameter and up probabilities. Each graph plots hit probabilities as a function of node storage. The top curve in each of these figures is an upper bound obtained from the techniques in Section 4. Each figure has a curve for $K = 1$ (Basic LRU Algorithm) and $K = 5$. The bottom curve is the hit probability for when the nodes do not cooperate. For the non-cooperative policy, when a node requests a file, it first checks its local storage to see if it has a cached copy; if not, the node retrieves the file from outside the community. For the non-cooperative policy, each node again uses LRU cache replacement. (Its local storage is set to the value of the shared storage in the corresponding cooperative algorithms.) The figure also includes curves for the MFR algorithm, which will be discussed shortly. We make the following observations:

- As we would intuitively expect, the hit probability increases if we increase the node storage capacity, Zipf

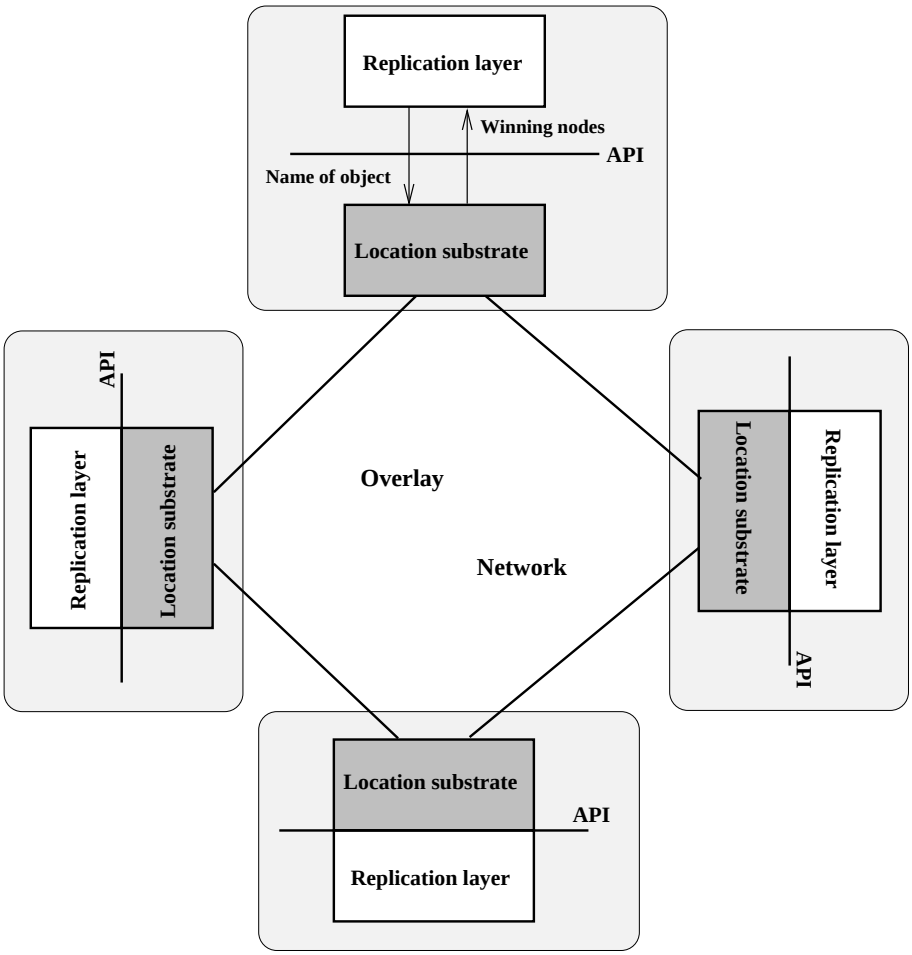


Figure 2: API

parameter, or the nodal up probability.

- The adaptive algorithm with $K = 1$ performs significantly better than the non-cooperative algorithm, but significantly worse than the theoretical optimal.
- Using a K value greater than 1 improves the hit probability, especially when nodes are frequently down. Further increasing K beyond $K = 5$ gives insignificant improvement. Figure 4 shows the fraction of misses for which the file was indeed available in some up node for the case Zipf parameter = 1.2 and $p = .2$.

Examining the number of replicas for each file provides important insight. Figure 5 shows, as a function of file popularity from most popular to least popular, the number of replicas per file for the theoretical optimal and for the adaptive LRU algorithm with $K = 1$. For the adaptive algorithm, the number of replicas per file is changing over time; the graphs therefore report the average values. The theoretical optimal number of replicas per file is obtained with the techniques in Section 4. Again, the figure presents four graphs, one for each of the combinations. The difference in how the theoretical optimal and the adaptive algorithm replicate files is striking. The optimal scheme replicates the more popular

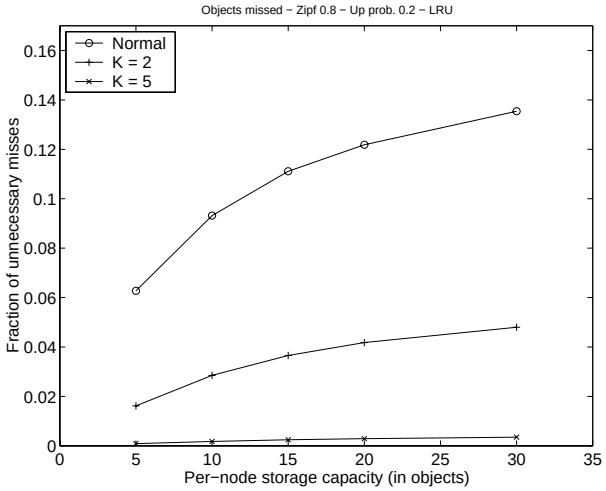
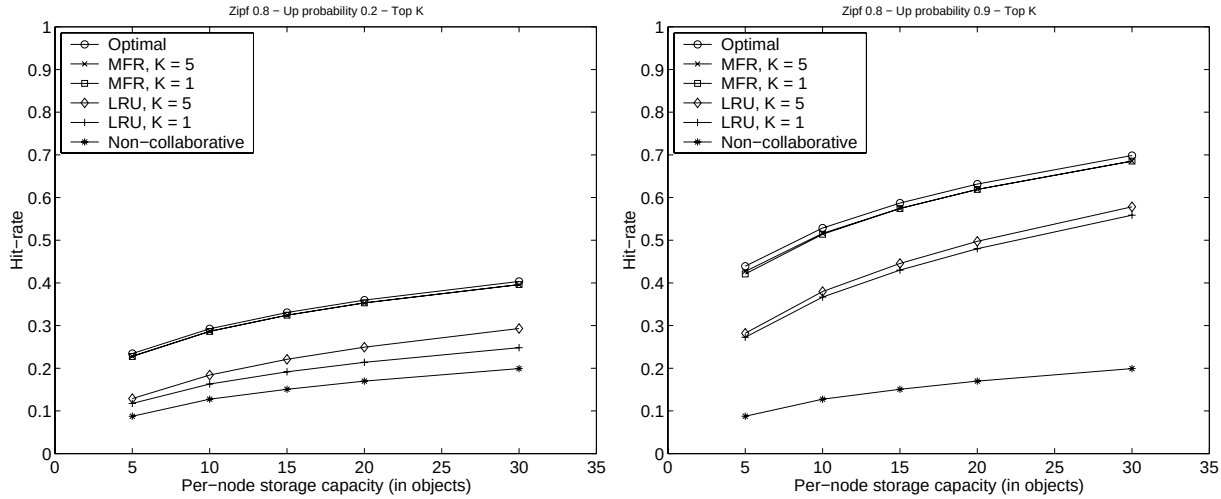
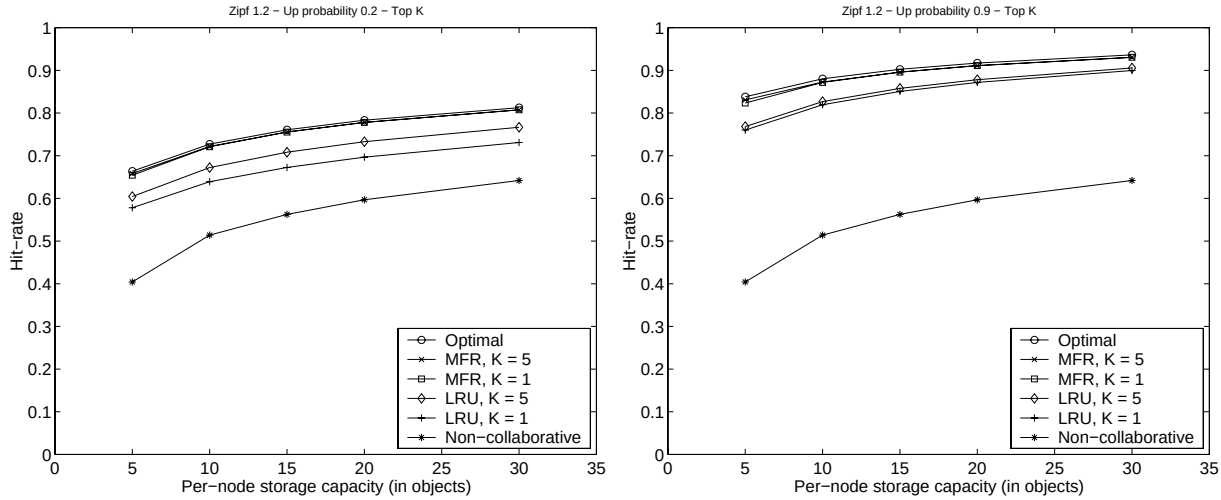


Figure 4: Fraction of misses for Zipf = 1.2

files much more aggressively than does the adaptive algorithm. Furthermore, the optimal scheme doesn't store the



(a) Zipf parameter .8, from left to right node up probabilities .2 and .9.



(b) Zipf parameter 1.2, from left to right node up probabilities .2 and .9.

Figure 3: Hit probability as function of node storage capacity

less popular files, whereas the adaptive algorithm provides temporary caching to the less popular files.

3.3 Top-K Most Frequently Requested Algorithm

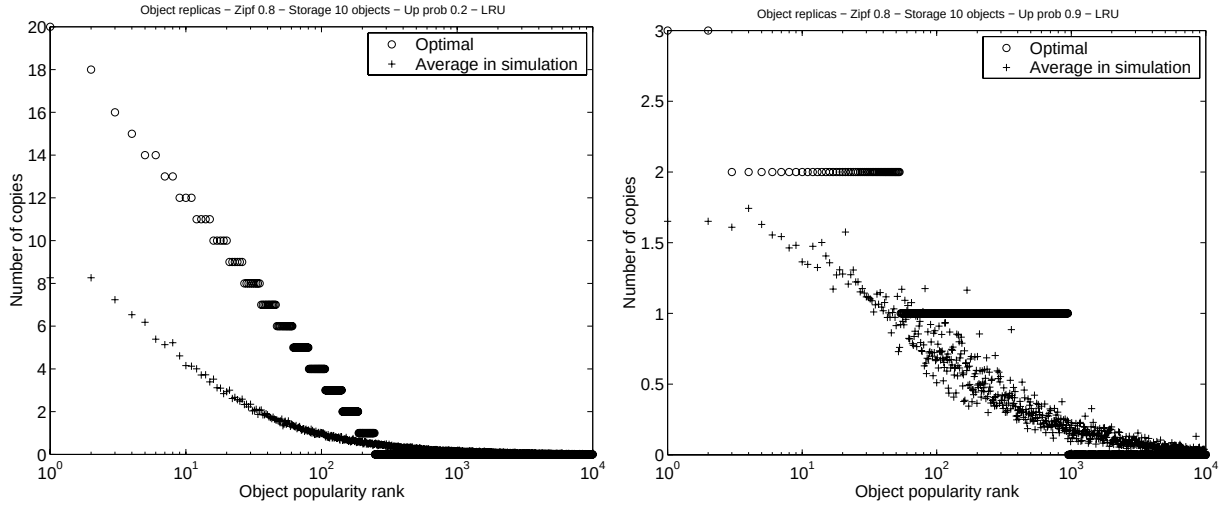
The Top-K LRU algorithm is simple and intuitive, but its performance is significantly below the theoretical optimal. We now consider how we can do better. To this end, we make the following two observations:

- LRU lets unpopular files linger in nodes. When an unpopular file is requested, it gets stored in one of the nodes and remains there until it is evicted with LRU. Intuitively, if we do not store the less popular files, the popular files will grab the vacated space and there will be more replicas of the popular files.
- Searching more than one node (that is, the top-K procedure) is needed to find files in the aggregate storage.

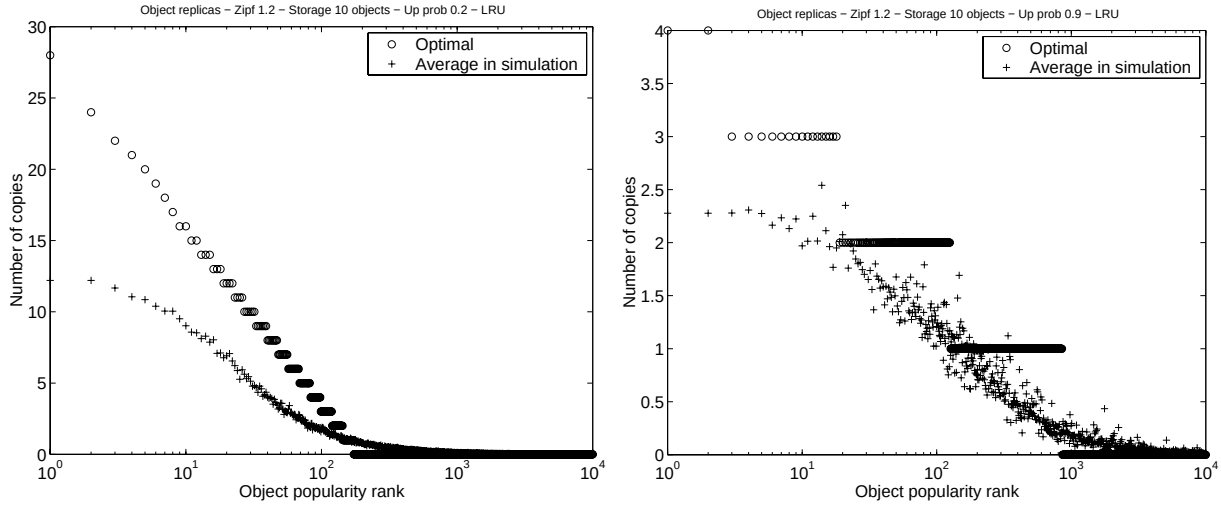
Based on these observations, we will now devise a new adaptive algorithm that has *near optimal performance*. To this end, we introduce the Most Frequently Requested (MFR) retrieval and replacement policy:

MFR retrieval and replacement policy

- Each node i maintains a table for all files for which it has received a request. For a file j in the table, the node maintains an estimate of $\lambda_j(i)$, the local request rate for the file. In the simplest form, $\lambda_j(i)$ is the number of requests node i has seen for file j divided by the amount of time node i has been up. In practice, we would likely weigh recent requests more heavily in the



(a) Zipf parameter .8, from left to right node up probabilities .2 and .9.



(b) Zipf parameter 1.2, from left to right node up probabilities .2 and .9.

Figure 5: Number of replicas per file with 10 files of per-node storage capacity and LRU replacement policy

online calculation of $\lambda_j(i)$. Also, note that ideally the table would contain an entry for *all* objects for which i has received a request; in practice the size of this table could be easily limited to, say, a few thousand most frequently requested objects without any impact on performance (recall that because of the large size of objects we are considering, a node would typically only be able to store a very small number of objects).

- Each node i stores the files with the highest $\lambda_j(i)$ values, packing in as many files as possible. As we show in Section 4, objects with different sizes should be ordered according to $\lambda_j(i)/b_j$, where b_j is the size of object j.

Thus when node i receives a request (from any other node) for file j, it updates $\lambda_j(i)$. It then checks to see if it currently has j in its storage. If i doesn't have j and MFR says it should[4], then i retrieves j from the outside, puts j in its storage, and possibly evicts one or more files from its storage according to MFR[5].

Now that we have defined the retrieval and replacement policy, we need to define the ping dynamics. We want the ping dynamics to influence the rates so that the numbers of replicas across all nodes become nearly optimal. One approach might be for X (the node that wants the file) to ping the top-K winners in parallel, and then retrieve the file from any node that has the file. Each of the pings could be con-

[4]If node i has storage for n objects, then i would store the n objects with the highest $\lambda_j(i)$ in its shared storage.

[5]There is a subtlety in how i gets j. The node i could simply retrieve i from the outside. The node i could also ping the remaining internal winners for the file, which only improves marginally the hit probability. (But in this latter approach, it is important not to count the pings as requests.)

sidered a request, and the nodes could update their request rates and manage their storage with MFR accordingly. But it turns out that this approach doesn't give better performance than Top-K LRU.

It turns out that the correct approach is for X to *sequentially* request j from the top-K winners, and stop the sequential requests once j is found. Sequential requests influence the locally-calculated request rates in a manner such that the global replication is nearly optimal. In particular the value of $\lambda_j(i)$ at any node i will be reduced (or "thinned") by hits at "upstream" higher-placed nodes for j. We now summarize the algorithm. Suppose X wants file j. Initialize $k = 1$.

Top-K MFR Algorithm

While $k \leq K$ and X has not obtained j:

1. X uses substrate to determine i, the kth place winner for j.
2. X requests j from i.
   - Node i updates $\lambda_j(i)$.
   - If node i already has j, node i sends j to X; stop.
   - If node i does not have j but it should (according to MFR), i gets j, stores j and evicts files if necessary. Node i sends j to X.
3. $k = k + 1$

If after K iterations, X still does not have j, X gets j from the outside directly (but does not put j in its shared storage). Note that asking the top-K winners sequentially will increase the delay of finding the object (or determining that it is not available in the community). However, since the objects are large, the download delay dominates the total delay experienced by the user and the delay to locate the object is only a minor fraction of the total delay.. In addition, downloads within the community happen at a much faster rate than from outside the community; hence, it pays off to ask the K winners within the community, even sequentially. In practice it is likely that the delay caused by contacting K peers in the community would be negligible and completely unnoticeable by the user.

Figure 6 shows, as a function of file popularity, the number of replicas per file for the theoretical optimal and for Top-K MFR Algorithm with $K = 5$. We see that, in contrast with LRU, the number of replicas given by the MFR algorithm is very close to the optimal. In fact for most files, the number of replicas given by the Top-5 MFR algorithm is equal to the optimal; a small fraction of files are off by one replica from the optimal. Figure 3 compares the hit rate of MFR (with $K = 1$ and $K = 5$) with the adaptive LRU algorithms and with the optimal hit rate. We see from Figure 3 that the MFR algorithms give hit rates that are very close to optimal over the entire parameter space considered. Again, we have observed similar results with heterogeneous file sizes, nodal storage capacities, and nodal up probabilities, and with smaller and larger Zipf parameters. The small and insignificant differences between MFR and optimal replication/replacement (when they occur) are due to imperfect load-balancing in the DHT substrate and to sub-optimal packing of non-constant-size files into the nodes' storage. *In conclusion, the Top-K MFR algorithm is a fully-distributed, adaptive content management algorithm that is, for all practical purposes, optimal for DHT-based file sharing systems.*

3.4 Performance Analysis of MFR

Given that the MFR algorithms possess many attractive properties, it is desirable to have available a performance evaluation technique for MFR that is more efficient than discrete-event simulation. We now present such a technique, which is not only accurate and efficient, but also sheds insight into the subtleties of the MFR algorithm.

To describe the performance evaluation technique, we introduce some additional notation. Let I denote the number of nodes. For a given node i, let p_i denote its up probability. Denote by S_i the amount of shared storage (in bytes) in the ith node. Let J denote the number of distinct files, and let b_j denote the size (in bytes) of the jth file. For this performance analysis, we assume that the request probabilities for the J files are known *a priori*. Specifically, we suppose that the request probability for each file j is a known value, q_j, with $q_1 + q_2 + \cdots + q_J = 1$.

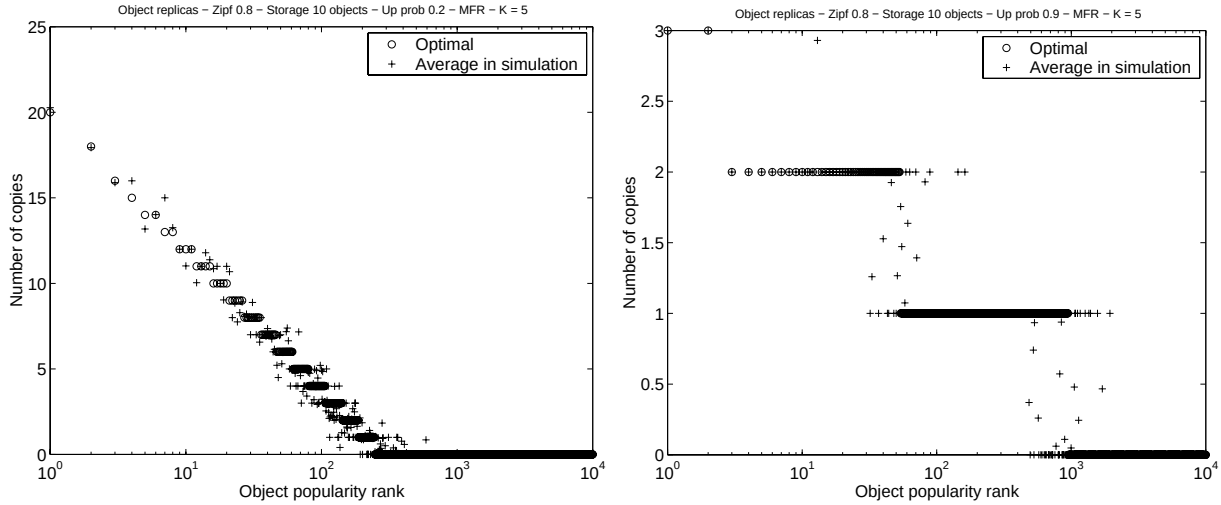
We now describe the analytical procedure for calculating the steady-state replica profile and hit probability for Top-K MFR for the case $K = I$. Although we only analyze the case $K = I$, the resulting replica profile and hit probabilities serve as excellent approximations for when K is small.

The procedure sequentially places copies of files into the nodes. Let T_i denote the remaining unallocated storage in node i; let x_{ij} be equal to 1 if a copy of file j has been placed in node i and equal to 0 otherwise. After placing a copy of file j in node i, T_i is reduced and x_{ij} is set to 1.

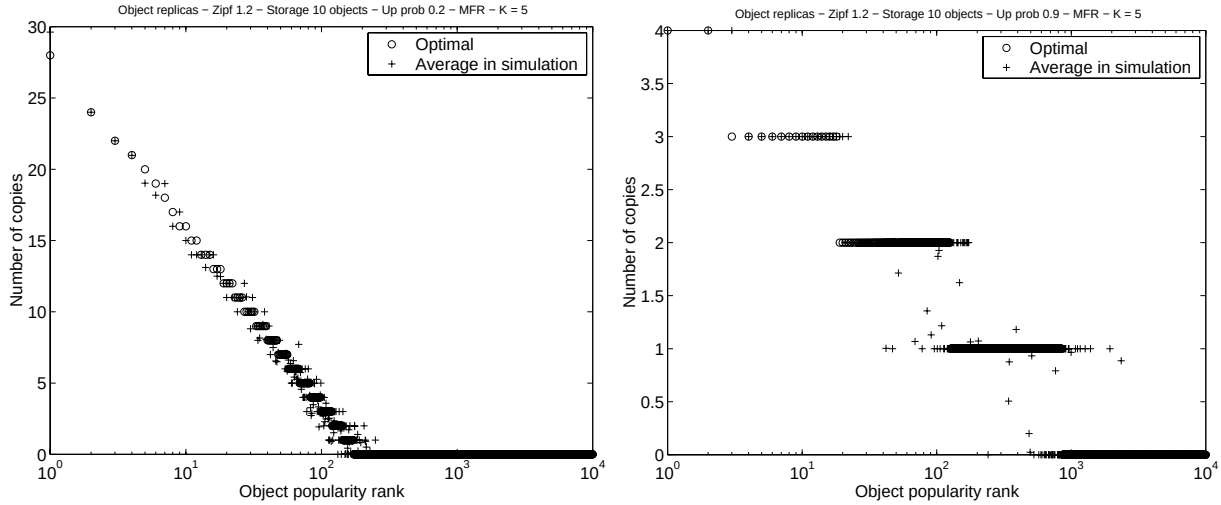
The procedure first initializes $\gamma_j = q_j/b_j$ for all $j = 1, \ldots, J$ and $T_i = S_i$ for all $i = 1, \ldots, I$. It also initializes $x_{ij} = 0$ for all $i = 1, \ldots, I$, $j = 1, \ldots, J$. At each iteration, the procedure chooses the file with the highest γ_j value, places a copy of that file in a node, and then reduces γ_j appropriately. Specifically,

1. Find the file j that has the largest value of γ_j.
2. Sequentially examine the winning nodes for j until a node is found such that $T_i \geq b_j$ and $x_{ij} = 0$.
   - Set $x_{ij} = 1$
   - Set $\gamma_j = \gamma_j(1 - p_i)$
   - Set $T_i = T_i - b_j$

   If there is no node such that $T_i \geq b_i$ and $x_{ij} = 0$, then remove file j from further consideration.
3. If all files have not been removed from consideration, return to Step 1. Otherwise, stop.



(a) Zipf parameter .8, from left to right node up probabilities .2 and .9.



(b) Zipf parameter 1.2, from left to right node up probabilities .2 and .9.

Figure 6: Number of replicas per file with 10 files of per-node storage capacity and MFR replacement policy with $K = 5$

The replication profile provided by this procedure is typically very close to the steady-state profile obtained by the Top-I MFR algorithm. The two profiles may differ slightly due to how files of different sizes are replaced and packed in the nodes, and due to ties in Step 1. However, under the conditions of the following theorem it can be shown that the two profiles are the same. (The proof of the theorem can be found in an extended version of the paper.) Let $\hat{x}_{ij}$, $i = 1, \ldots, I$, $j = 1, \ldots, J$ be the final x_{ij} values from the above procedure. Let $x_{ij}(t)$ be equal to one if at time t there is a copy of file j in node i when the Top-I MFR algorithm is used.

Theorem 1: Suppose all files are the same size. Further suppose that there are never any ties in Step 1 of the procedure. Then $x_{ij}(t)$ almost surely converges to $\hat{x}_{ij}$ for all $i = 1, \ldots, I$ and $j = 1, \ldots, J$.

An interpretation of the Theorem 1 is as follows. Note that γ_j is proportional to the request rate to the outside for file j. Whenever a copy of file j is put in node i, the external request rate for file j is 0 when node i is up and is not reduced when node i is down; thus the expected external request rate is reduced to $\gamma_j(1 - p_i)$. Since Top-K MFR converges to replica profile of the procedure (Theorem 1), the adaptive algorithm has the effect of giving priority to files that have the highest thinned external request rates.

We used this procedure and Theorem 1 to evaluate the performance of the Top-I MFR algorithm. Specifically, we ran the procedure for 30 test cases, with the different cases obtained by different combinations of the nodal up probability, Zipf parameter, and nodal storage capacity. Each case had 100 nodes. For the same cases, we obtained the theoretical optimal solution (see Section 4). For all of the cases, the

Table 1: Request probabilities and winners

file	Req. prob.	1st place winner
1	5/13	1
2	3/13	2
3	3/13	2
4	2/13	1

conditions of Theorem 1 were satisfied; thus, the procedure provides the replication profile for Top-I MFR algorithm in steady state. We found that in 28 of the 30 cases the MFR algorithm (with $K = I$) converges the optimal replication profile! In the two cases without convergence, the number of replicas were the same for all but two files, with one more replica for one of the files and one less for the other. This experiment further confirms that the MFR algorithms are near optimal.

We now provide a simple example for which the MFR-I algorithm does not produce the optimal replication profile. This example gives insight into why MFR-I is not always optimal. This example has $I = 2$ nodes, each capable of storing two files, each having up probability 0.5. It has four file types, with request probabilities and first-place winners shown in Table 1. It is easily seen that the Top-2 MFR algorithm puts one copy of file 1 and one copy of file 4 in node 1, and one copy of file 2 and one copy of file 3 in node two. However, the optimal solution puts one copy of file 1 and one copy of file 3 in the first node, and one copy of file 1 and one copy of file 2 in the second node. The problem comes from the fact that MFR first assigns files 1, 2 and 3, thereby filling node 2. For the next assignment, file 1 is more desirable than file 4; however, file 1 cannot be assigned to node 1, as there is already a copy of file 1 there.

3.5 Hot Spots

Up until this point our focus has been on managing content to maximize the probability of having a hit in the community. However, consider the case when a very popular object j has a first-place winner i that is almost always up. In this case, the adaptive algorithms will only create one copy of object j, which will be permanently stored on peer i. If the demand for this object is very high, then peer i will become overloaded with file transfers. In this section we sketch approaches to solving the hot-spot problem.

One approach to this problem is to segment all files (or just popular files) into multiple fragments, and give each fragment a unique name. Each fragment is then treated as a separate file in the Top-K MFR algorithm; thus the file-transfer load imposed by a popular object becomes spread over many nodes. One drawback to this approach is that, with multiple fragments per file, a hit requires having a hit for each of the individual fragments. A further refinement of the approach is to use erasures, that is, to create R erasures for the popular files in a manner such that the original file can be reconstructed from any M of the R erasures. In the following section we provide an upper bound on the performance of adaptive schemes that use erasures.

Another approach to this problem is to leave the files intact (no fragmentation) and allow nodes overloaded with file transfers to reject requests even when they have a copy of the requested file. Thus, if the first-place node for a particular file is overloaded, it sends a negative message back to requesting node. The requesting node then requests the file from the second-place winner, and so forth. We refer to this approach as the *overflow approach*. Adaptive algorithms using erasures and using overflows is the subject of a subsequent paper.

Of course, both the fragmentation and the overflow approaches to hot spot problems only help to relieve file-transfer loads; they do not reduce the number of requests to the top winners of a popular file. We are also currently investigating algorithms for which the request load for a popular file is spread over multiple nodes.

4. OPTIMIZATION THEORY

We now complement our suite of adaptive algorithms with an analytical theory for optimal replication in P2P communities. The theory applies to both open communities (as studied in the previous sections) and closed communities, for which it is not possible to access files from outside the community. We also derive the theory in a more general context for which each file (is possibly) erasure coded. In a subsequent paper, we will study adaptive algorithms for erasure encoded files in open and closed communities. Here our focus is on obtaining upper bounds.

For this analytical theory, we assume that file popularities and nodal up probabilities are known *a priori*. As in Section 3.4, let I denote the number of nodes, and let p_i and S_i denote the up probability and shared storage for node i. In this section we suppose that nodes go up and down independently. Let J denote the number of distinct files, and let b_j and q_j denote the size and request probability of the jth file.

4.1 No Fragmentation

We first develop the theory for the case when files are not fragmented (the case considered in the previous sections). Let x_{ij} be a zero-one variable which is equal to one if node i contains a replica of file j and is zero otherwise. It is straightforward to show that the hit probability is given by

$$P_{hit} = 1 - \sum_{j=1}^{J} q_j \prod_{i=1}^{I} (1 - p_i)^{x_{ij}}. \quad (1)$$

The assignments must satisfy the constraints

$$\sum_{j=1}^{J} b_j x_{ij} \leq S_i, \quad i = 1, \ldots, I \quad (2)$$

The solution of the integer programming problem of maximizing (1) subject to (2) provides an upper bound on the hit probability for all content management algorithms. The integer program can thus be used to benchmark adaptive algorithms. However, this optimization problem can be shown to be NP-complete by reducing it to the Zero-One Integer Programming problem [19].

We now consider a special case of this problem, namely,

when each node is up with the same probability $p_i = p$. Let n_j denote the number of replicas for file j. For the case of homogeneous up probabilities, the problem is to choose non-negative integers $n_1, \ldots, n_J$ such that the following is maximized

$$1 - \sum_{j=1}^{J} q_j (1-p)^{n_j} \tag{3}$$

subject to

$$\sum_{j=1}^{J} b_j n_j \leq S \tag{4}$$

where $S = S_1 + \cdots + S_I$. Note that, because storage has been aggregated into one constraint, the optimal value for (3-4) is actually an upper bound on the optimal hit probability (for homogeneous values of p_i); however, this upper bound turns out to be very tight, and can actually shown to be exact for important special cases. (For example, when all files are the same size.) This optimization problem can be solved efficiently by dynamic programming. Indeed, let $f_j(s)$ be the minimum miss probability when there are s bytes of aggregate storage and files $j, \ldots, J$ to replicate. Then standard dynamic programming arguments give

$$f_j(s) = \min_{n: b_j n \leq s} [f_{j+1}(s - nb_j) + q_j(1-p)^n]$$

The optimal replica profile $n_1, \ldots, n_J$ solves $f_1(S)$. This provides the "theoretical optimal" used in the Section 3 to benchmark our adaptive algorithms.

We note in passing that the same methodology can be used to obtain an upper bound when there are two heterogeneous up probabilities, one for each of the two sets in a two-set partition of the nodes. (For example, 20% of the nodes up with probability .9 and 80% of the nodes up with probability .2.) In this case, we can derive a two-dimensional dynamic programming equation. The computational complexity increases, but the computations remain tractable for large values of J.

4.2 Upper Bound With and Without Erasures

The number of copies of any of object in the P2P community is an integer at any given time. By removing this integrality restriction, we can develop a methodology for efficiently determining an upper bound on the performance of adaptive management algorithms in large P2P systems. We shall do this for the case of erasures; the case without erasures will be treated as a special case.

We now suppose that each file j is made up of R_j erasures, and that any M_j of the R_j erasures are needed to reconstruct the file. The size of each erasure is b_j/M_j. Throughout this analysis we assume homogenous up probabilities, that is, $p_i = p$ for all nodes. We also make the natural restriction that the replication algorithms that we are bounding are such that no two erasures from the same file are stored on the same node. Let $c_j = M_j/(b_j R_j)$ and

$$f_j(z) = q_j \sum_{m=M_j}^{R_j} \binom{R_j}{m} [1-(1-p)^{c_j z}]^m [(1-p)^{c_j z}]^{R_j - m}$$

The main result of this subsection is the following:

Theorem 2: The maximum value of the following optimization problem provides an upper bound on the hit probability for a P2P file-sharing community with erasures.

Maximize

$$\sum_{j=1}^{J} f_j(z_j) \tag{5}$$

subject to

$$\sum_{j=1}^{J} z_j = S \tag{6}$$

$$z_j \geq 0, \qquad j = 1, \ldots, J \tag{7}$$

The optimization problem in Theorem 2 is easy to solve numerically, even for large values of J and R_j, $j = 1, \ldots, J$. Specifically, it is straightforward to show that $f_j(z)$ is an increasing concave function of z, so that optimization problem is a separable concave allocation problem. Let $(z_1^*, \ldots, z_J^*)$ be the optimal solution for this concave optimization problem. From Kuhn-Tucker theory, there exists an α such that

$$f_j'(z_j^*) = \alpha \text{ for all } j \text{ such that } z_j^* > 0 \tag{8}$$

The standard procedure to solve this type of problem is to first pick an $\alpha > 0$, solve $f_j'(z_j) = \alpha$ for all j; for those values of j such that $z_j \geq 0$, we sum the z_j's and check if the sum is above or below S. If the sum is above S (below S) we decrease α (increase α) and repeat the procedure. Using a binary search to adjust α, we iterate until the sum of the positive z_j's is within ϵ of S. For the positive z_j's we set $z_j^* = z_j$; for the remaining z_j's we set $z_j^* = 0$.

Theorem 2 provides a powerful means to benchmark the performance of adaptive algorithms with and without erasures. We provide an example at the end of this section that shows that the upper bound is quite tight when entire files are replicated.

Proof of Theorem 2: We refer to the rth erasure of file j as erasure jr, $r = 1, \ldots, R_j$. For a fixed assignment of erasure replicas to nodes, let n_{jr} be the number of erasures jr stored in the community of nodes. Clearly

$$\sum_{j=1}^{J} \sum_{r=1}^{R_j} \frac{b_j}{M_j} n_{jr} \leq S \tag{9}$$

Using the same fixed assignment, let Φ_{jr} be the 0-1 random variable which is 1 if any of the n_{jr} erasures jr is in some up node. Clearly,

$$P(\Phi_{jr} = 1) = 1 - (1-p)^{n_{jr}} := p_{jr} \tag{10}$$

Let $P_j(hit) = P(\text{hit}|\text{request for } j)$. Because there is a hit for a request for file j if any M_j of the R_j erasures for file j are available, we have

$$P_j(hit) = \sum_{m=M_j}^{R_j} P(\sum_r \Phi_{jr} = m). \tag{11}$$

Let $\mathcal{R}_j(m) = \{A : A \subseteq \{1,\ldots,R_j\} \text{ and } |A| = m\}$. We also have

$$\begin{aligned} P(\sum_r \Phi_{jr} = m) &= \sum_{A\in\mathcal{R}_j(m)} P(\cap_{r\in A}\Phi_r \cap_{r\in A^c} \Phi_r^c) \\ &= \sum_{A\in\mathcal{R}_j(m)} (\prod_{r\in A} p_{jr})(\prod_{r\in A^c}(1-p_{jr})) \end{aligned} \tag{12}$$

where the last inequality follows from the fact that no two erasures from the same file are stored on the same node. Let $\mathcal{T}_j = \{A : A \subseteq \{1,\ldots,R_j\} \text{ and } |A| \geq M_j\}$. Combining (10-12) gives

$$\begin{aligned} P_j(hit) &= \sum_{A\in\mathcal{T}_j}\prod_{r\in A}[1-(1-p)^{n_{jr}}]\prod_{r\in A^c}[(1-p)^{n_{jr}}] \\ &:= h(n_{j1}, n_{j2}, \ldots, n_{jR_j}) \end{aligned} \tag{13}$$

Now consider a reliability system that consists of R_j subsystems, with the rth such subsystem consisting of n_{jr} parallel components, with every component being operational with probability p. Suppose that the system is operational if any M_j out of R_j subsystems are operational. It is easy to see that the probability that the system is operational is given by $h(n_{j1}, n_{j2}, \ldots, n_{jR_j})$. Combining this observation with Theorem 2.2 of Boland et al [24] implies that $h(n_{j1}, n_{j2}, \ldots, n_{jR_j})$ is Shur concave. This in turn implies that this $h(n_{j1}, n_{j2}, \ldots, n_{jR_j}) \leq h(x_j, x_j, \ldots, x_j)$, where $x_j = (1/R_j)/\sum_{r=1}^{R_j} n_{jr}$. Thus

$$\begin{aligned} &\sum_{A\in\mathcal{T}_j}\prod_{r\in A}[1-(1-p)^{n_{jr}}]\prod_{r\in A^c}[(1-p)^{n_{jr}}] \leq \\ &\qquad \sum_{A\in\mathcal{T}_j}[1-(1-p)^{x_j}]^{|A|}[(1-p)^{x_j}]^{|A^c|} \end{aligned} \tag{14}$$

Combining (13) and (14) gives

$$P_j(hit) \leq \sum_{m=M_j}^{R_j} \left(\begin{array}{c} R_j \\ m \end{array} \right) [1-(1-p)^{x_j}]^m [(1-p)^{x_j}]^{R_j-m} \tag{15}$$

Now define $z_j = x_j/c_j$. From (15) and the definition of $f_j(z)$ we have

$$P(hit) \leq \sum_{j=1}^{J} f_j(z_j) \tag{16}$$

and from (9)

$$\sum_{j=1}^{J} z_j \leq S. \tag{17}$$

The result follows directly from (16-17). □

4.3 Logarithmic Replication Rule

Theorem 2 of the previous section can be used to benchmark adaptive algorithms that use erasures. Moreover, by specializing the theorem to adaptive algorithms without erasures (as studied in Section 3), we can obtain a closed-form expression for the optimal hit probability, which sheds additional insight in on how files should be optimally replicated.

First we note that for the case of no erasures, we have $R_j = M_j = 1$, so that $f_j(z)$ simply becomes $f_j(z) = q_j(1-p)^{z/b_j}$ for all $j = 1,\ldots,J$. Let $(z_1^*, z_2^*, \ldots, z_J^*)$ be the optimal solution to the optimization problem in Theorem 2. This optimal solution can be obtained explicitly as follows. Differentiate $f_j(z)$ and solve for z_j^* such that $f_j(z_j^*)\alpha$, and use (6) to solve for α, and taking special care that $z_j^* \geq 0$ is not violated, we obtain the following solution. Now reorder the files so that they have decreasing valuses of q_j/b_j. There is an L such that $z_j^* = 0$ if and only if $j > L$. (We will indicate shortly how L is determined.) Define

$$B_L = \sum_{j=1}^{L} b_j$$

which is the amount of storage required by the first L files. Finally, let $n_j^* = z_j^*/b_j$, which has the interpretation of the optimal number of replicas of file j in the continuous relaxation of the problem. After carrying out this exercise, we obtain:

$$n_j^* = \frac{S}{B_L} + \frac{\sum_{l=1}^{L} b_l \ln(q_l/b_l)}{B_L \ln(1-p)} + \frac{\ln(q_j/b_j)}{\ln(1/(1-p))} \tag{18}$$

It remains to specify how L is determined. This is done by finding the largest L such that $n_L^* > 0$ using (18). This can be done by a simple (linear or binary) search.

The n_j^* given by (18) is non-integer and represents the approximate number of replicas for file j. When these values are used in the expression for $P(hit)$, we obtain a closed-form approximation for the hit probability:

$$P_{approx}(hit) = 1 - a^{S/B_L}\sum_{j=1}^{L} q_j \prod_{l=1}^{L}(\frac{q_l/b_l}{q_j/b_j})^{b_l/B_L} \tag{19}$$

where $a = 1-p$.

The expressions (18-19) provide significant insight into the nature of the optimal replica profile:

- The ratio q_j/b_j plays a key role in influencing the number of replicas that are assigned to object j. Objects with small values of q_j/b_j (specifically, for objects with $j > L$) are not stored in any of the peers in the optimal solution.
- We call the replication of objects given by (18) the *logarithmic replication rule* since n_j^* is equal to a constant plus a term proportional to $\ln(q_j/b_j)$. It is interesting to note a parallel of the logarithmic replication rule with the *square root assignment rule* that was derived by Kleinrock for the link capacity assignment problem in 1964 [20]!
- The expression (19) is actually upper bound on the true optimal hit rate, since it is the optimal over continuous variables rather than integer variables.

We now provide some numerical results. Table 2 shows the difference between (19) and (3) as a percentage of the upper bound given by (19). We show the results for three different node up probabilities and two different values of the Zipf parameter. In all cases, we had 100 nodes and each node had storage capacity for 15 objects (all objects assumed to be the same size). We observed similar behavior for other

Table 2: Average difference between continuous solution and dynamic programming solution

Up prob.	Zipf 1.2	Zipf 0.8
0.2	0.02%	0.08%
0.5	0.1%	0.7%
0.9	0.9%	5.8%

storage capacities not reported here. The results in Table 2 show that in most cases the upper bound given by (19) is very tight; typically the difference is less than 1%. However, as the up probability goes up and the Zipf parameter goes down, the difference increases somewhat. This behavior is understood by comparing the optimal number of replicas obtained from the logarithmic replication rule with that obtained from dynamic programming; see Figure 7.

We can clearly see the logarithmic replication rule in Figure 7. The replicas given by (18) decrease logarithmically as the file popularity decreases. The replicas given by the dynamic programming are constrained to integer values and decrease in steps, closely following the continuous optimal in most cases. In cases where the most popular objects need only a few copies, such as the case shown in 7(b), the optimal number of integer replicas is coarse. This explains the large difference in hit-rates shown in Table 2.

5. CONCLUSIONS AND FUTURE WORK

P2P file sharing (e.g., KaZaA and Gnutella) is an enormously popular Internet application and accounts for the majority of today's Internet traffic. Although today the popular file sharing applications are "unstructured" designs, structured, DHT-designs will potentially improve search and download performance.

One of the features of structured, DHT-based P2P file sharing is that the application has significant control on where and how many replicas are generated. The contribution of this paper is twofold.

- First, for DHT-based file sharing systems, we have proposed a suite of adaptive algorithms for replicating and replacing files as a function of evolving file popularity. In particular, we proposed the Top-K MFR algorithm, which is a fully-distributed, adaptive, near-optimal content management algorithm for DHT-based file sharing systems.
- Second, we have introduced an optimization methodology for benchmarking the performance of adaptive management algorithms. The methodology directly applies to networks whose nodes have homogeneous up probabilities, and can be extended to heterogeneous environments (e.g., two or three different sets of nodes, each with their own up probability). The methodology applies to designs that use erasures; it also applies to closed P2P communities as well as open communities.

The algorithms and theory developed in this paper set the stage for several other important problems. We are currently developing new adaptive algorithms for closed P2P communities, P2P communities that use erasures, and P2P communities that use request overflow to abate hot spot problems.

6. REFERENCES

[1] KaZaA, http://www.kazaa.com

[2] Gnutella, http://www.gnutella.com

[3] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, Wide-area cooperative storage with CFS, *ACM SOSP 2001*, Banff, October 2001

[4] P. Druschel and A. Rowstron, "PAST: A large-scale, persistent peer-to-peer storage utility", *HotOS VIII*, , May 2001

[5] J. Kubiatowicz et al. "OceanStore: An Architecture for Global-Scale Persistent Storage," *ASPLOS 2000*, Nov. 2000.

[6] I. Stoica, D. Adkins, S. Zhaung, S. Shenker, and S. Surana, "Internet Indirection Infrastructure," *Proceedings of ACM SIGCOMM'02*, Pittsburgh, August 2002.

[7] M. Castro, P. Druschel, A-M. Kermarrec and A. Rowstron, "SCRIBE: A large-scale and decentralised application-level multicast infrastructure", *IEEE Journal on Selected Areas in Communications (JSAC)* (Special issue on Network Support for Multicast Communications). 2002, to appear

[8] S. Iyer, A. Rowstron and P. Druschel, "SQUIRREL: A Decentralized, Peer-to-Peer Web Cache", *PODC 2002*.

[9] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed file location and routing for large-scale peer-to-peer systems". *IFIP/ACM International Conference on Distributed Systems Platforms*, Heidelberg, Germany, Nov. 2001

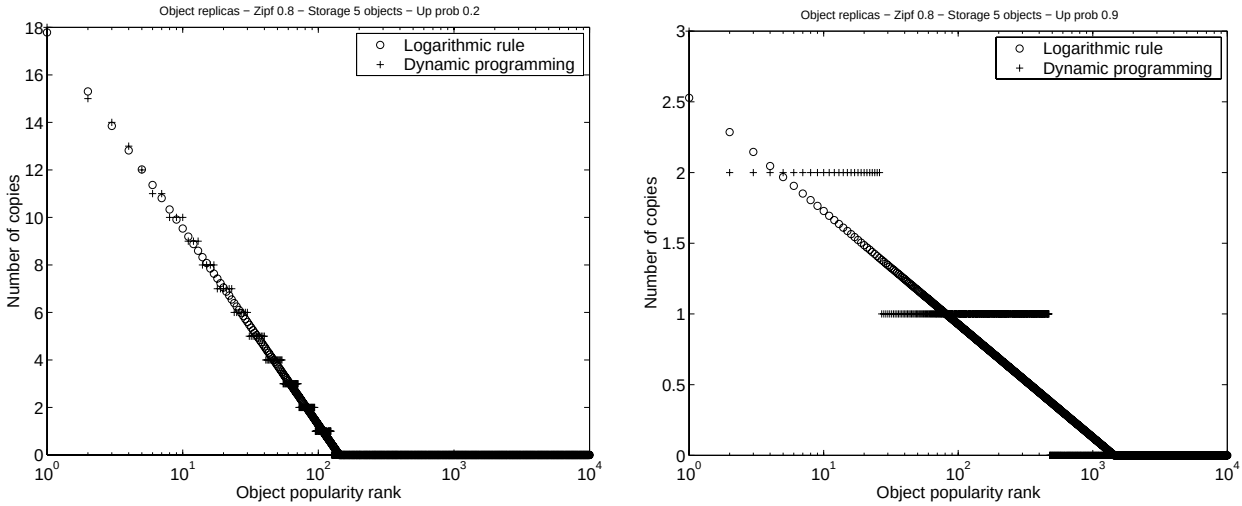
[10] H. Weatherspoon and J. D. Kubiatowicz, "Erasure Coding vs. Replication: A Quantitative Comparison," *IPTPS '02*.

[11] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *SIGCOMM 2001*, San Diego, CA.

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

[13] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker, "Search and replication in unstructured peer-to-peer networks," in *ACM International Conference on Supercomputing*, June 2002.

[14] E. Cohen and S. Shenker, "Replication strategies in unstructured peer-to-peer networks," in *SIGCOMM 2002*.



(a) Up prob. = .2 (b) Up prob. = .9

Figure 7: Values of n_j and n_j^* for Zipf .8 and 5 objects of per-node storage

[15] Chronicle of Higher Education, "Napster was nothing compared with this year's bandwidth problems," Sept. 28, 2001.

[16] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph, "Tapestry: An infrastructure for fault-tolerant wide-area location and routing," Tech. Rep. UCB//CSD-01-1141, Apr. 2000.

[17] L. Breslau, P. Cao, G. Phillips, and S. Shenker, "Web caching and Zipf-like distributions: Evidence and implications," in *INFOCOM 1999*.

[18] K. Sripanidkulchai, "The popularity of Gnutella queries and its implications on scalability," Mar. 2001, Unpublished.

[19] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, 1979.

[20] L. Kleinrock, *Queueing Systems. Volume II: Computer Applications*, John Wiley & Sons, New York, 1976.

[21] A. Adya et al. "FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment," in *OSDI'02*, December 2002

[22] J.R Douceur, A. Adya, W.J. Bolosky, D. Simon, M. Theimer, "Reclaiming Space from Duplicate Files in a Serverless Distributed File System," in *ICDCS, 2002*

[23] J.R. Douceur, R.P. Wattenhofer, "Optimizing File Availability in a Secure Serverless Distributed File System," Proceedings of *20th IEEE SRDS, 2001*

[24] P.J Boland, E. El-Neweihi, F. Proschan "Stochastic Order for Redundancy Allocations in Series and Parallel Systems," *Advances in Applied Probability*, 1992, pages 161-71.