Adaptive Content Management in Structured P2P Communities

Jussi Kangasharju* Keith W. Ross† David A. Turner‡

*Dept. of Computer Science
TU Darmstadt
Darmstadt, Germany

†Dept. of Computer and Information Science
Polytechnic University
Brooklyn, NY

‡ Dept. of Computer Science
California State University
San Bernardino, CA

***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) overlay is available for locating the node responsible for a key. In particular, we develop the Top-K MFR replication and replacement algorithm, which can be layered on top of a DHT overlay, and in addition adaptively converges to a nearly-optimal replication profile. Furthermore, we evaluate the file transfer load caused by the adaptive algorithms on each peer, and present two approaches for achieving a better load balance. Our evaluation shows that with our two algorithms, an arbitrary load distribution is possible, hence allowing each peer to serve requests at the rate it wishes.**

I. Introduction

One of the most popular uses of the Internet today is P2P file sharing of multimedia content. Popular P2P file sharing systems, such as eDonkey, Gnutella2, and FastTrack, 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 increasingly more and more 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 [3].

The file sharing systems Gnutella and FastTrack 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, such P2P systems must use a limited-scope search for finding content, which might be unsuccessful, even if the content is available. Furthermore, the unstructured file sharing systems do not attempt to replicate content in a manner that is socially advantageous for the P2P community at large.

In contrast to the "unstructured" P2P systems, structured P2P systems, such as CAN [12], Chord [13], Pastry [10], and Tapestry [16], use *distributed hash table (DHT) substrates*, which organize nodes into highly-structured overlay networks and which deterministically assign keys to nodes. A DHT overlay can serve as a platform for a variety of P2P applications, including persistent file storage [4]–[6], multicast [7], [8], mobility management [7], and Web caching [9]. DHT overlays 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.

We address the problem of content management in P2P file-sharing communities. The content management problem consists of deciding how many copies of each object should be stored in the community and on which nodes in the community should those copies be placed. Our focus is in on the sharing of large audio and video files in well-connected communities, such as campuses. We assume the nodes in the community are part of a DHT-based overlay network. Throughout, we make the natural assumption that intra-community file transfers occur at relatively fast rates as compared with file transfers into the community.

The essence of our problem is to adaptively manage content in a P2P community to minimize the average download delay, which we define 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 IV.)

The main challenge in the content management problem lies in designing an algorithm that is simple, decentralized, adaptive, and converges to the optimal content placement. The problem of designing such a content management algorithm consists of three important issues:

- **Replication:** Because nodes connect to and disconnect from the community, content needs to be replicated in the community to provide satisfactory hit rates. Naturally, popular content needs more replicas than unpopular content. At the same time, content should not be excessively replicated, wasting bandwidth and storage resources.

- **File Replacement Policies:** Each participating node has a limited amount of storage that it can offer to the community. When this storage fills at some node, the node needs to determine which files it should keep and which it should evict.
- **Load Balancing:** Because some files are more popular than others, nodes storing these files can encounter a much higher load than nodes storing less popular content. Load balancing mechanisms are needed to ensure that the load is balanced as evenly as possible across all the nodes.

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 overlays (e.g., [10], [12], [13], [16]).

We devise an algorithm, called *Top-K Most Frequently Requested Replication Algorithm* (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 evaluation of the load balancing properties of our adaptive algorithms. As our evaluation shows, the load caused by the basic algorithms can be heavily concentrated on a few nodes, we develop two additional load balancing mechanisms: fragmentation and overflow approach. The fragmentation approach allows us to perform a general load balancing, thus making the load uniform across all the peers. The overflow approach allows individual peers to deviate from this "fair share" if they do not have enough resources. Our evaluation shows that the two mechanisms together allow us to reach a well-balanced load with little adverse effects.

This paper is organized as follows. Section II reviews related work. Section III discusses structured P2P communities. In Section IV we propose and analyze distributed content management algorithms, including Top-K Replication Algorithm and Top-K Most Frequently Requested Replication Algorithm. In Section V we evaluate the load balancing properties of our algorithms and propose two algorithms for balancing the load. In Section VI we conclude and discuss future directions.

II. Related Work

A P2P community can also be viewed as a distributed P2P cache for caching large multimedia files. Squirrel [9] is a recent proposal and implementation of a distributed, serverless, P2P Web caching system. Squirrel, which is built on top of the Pastry [10] DHT overlay, 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 [9] has not focused on the fundamental issues of replication and file replacement in a P2P community. Furthermore, the focus of [9] 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 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 a file depends on the popularity of the file.

Lv et al [14] and Cohen and Shenker [15] 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.

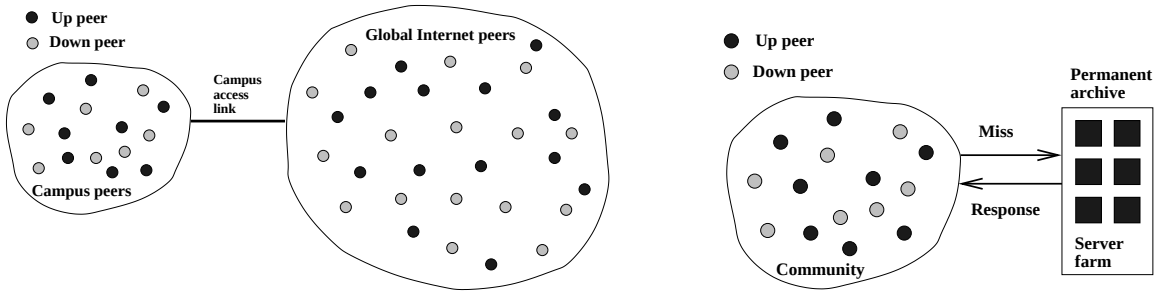
There has also been work comparing replication and erasure coding in distributed storage infrastructures [11]. The authors also discuss the drawbacks of using erasures. The paper focuses on persistent storage, and does not consider adaptive replication and replacement of replicas or erasures.

III. Structured Communities

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.

Note that the main goal of bringing the peers together in the community is to reduce traffic on the network links connecting the community to the outside. As such, even home users in one ISP could form a community, since that would reduce traffic



(a) Campus community

(b) Content distribution booster

Fig. 1. Different types of P2P communities

on the ISP's outgoing link. Although the connection to another home user in the same ISP might not be significantly faster than a connection to any other peer in the Internet, reducing traffic on the outgoing link will speed up the downloads which must be retrieved from outside of the community. Hence, the users obtain a better quality of service than they would without the community.

A. DHT Overlays

We assume that each community is built on a DHT overlay. The DHT only covers the peers in the community and different communities could use different DHTs.

Each node has a persistent identifier, which is assigned when the node initially subscribes to the application (assigned by the DHT). We assume that each node has access to a DHT substrate which 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. Any current DHT substrate, like CAN, Chord, Pastry, or Tapestry, already provides the first-place winners and can easily be extended to provide the top K winners.

IV. Adaptive Algorithms for Content Management in P2P Communities

As discussed in Section I, a P2P community is 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.[1] Each participating node allocates some *shared storage* to the P2P community. (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, as is generally the case in P2P file-sharing systems.

As mentioned above, a natural performance measure is *average delay*, where delay is defined as the delay to locate and to download a file. Because our focus is on large files, 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[2].

A. Optimal Replication Algorithms

In our previous work [25], [26], we have developed an analytical optimization theory for replication in P2P communities. This replication theory allows us to determine the optimal hit-rates and replica profiles both for complete-file replication as well as for segmented-file replication in which erasure codes are used to provide redundancy. For the general case with erasures, we have derived an upper bound on the performance of all adaptive schemes. For complete-file replication, we have shown that an explicit *logarithmic assignment rule* is optimal.

This optimization theory allows us to benchmark our adaptive algorithms. As mentioned in [25], [26], the optimization problem is NP-complete, but in the case where the peer up probabilities are homogeneous, the problem can be solved with dynamic programming. Hence, in the following we will concentrate on this case, since it allows us to benchmark our adaptive algorithms and evaluate how close to the optimal policy they are.

B. Top-K Replication 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 overlay 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 overlay to search for a replica in the community; if the community does not find a replica, 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

[1] Low-resource devices would not likely cache and serve files, but might be permitted to download files.

[2] The algorithms developed in this paper are easily modified also for byte hit performance.

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 Replication Algorithm

1) X uses the overlay 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 Replication 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 Replication Algorithm will retrieve file j from the outside even though it is cached in the community. Suppose that j is cached at its first-place winner and prior to the next request, a new node comes up and becomes the new first-place winner. The Basic Replication Algorithm would then retrieve j from the outside. To mitigate this problem, we modify the Basic Replication 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, i_1 retrieves j from the outside. We refer to this modified algorithm as the **Top-K Replication Algorithm**.

Observe that the Top-K Replication Algorithm replicates content without any *a priori* knowledge of file request patterns or node up probabilities, and is 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 a miss is negligibly small. In the Top-K Replication algorithm, if i_1 finds file j in any of the nodes $i_2, \ldots, i_K$, it is better for i_1 to copy j instead of simply storing a pointer to it. This is because i_1, as the first place winner, will receive all requests for j until it goes down (or a new first place winner emerges). If i_1 only has a pointer to the node with j and that node goes down, file j may be lost to the community. If there are several nodes which have a copy of the object, then i_1 could defer the copying until only a small number of copies remain, however, this would mean increased ping traffic since i_1 would have to ping all the other winners for every single request.

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[3].

As mentioned in Section IV-A, the theoretical optimal replication policy and corresponding upper bound are feasible to calculate only when the node up probabilities are homogeneous, i.e., all nodes behave the same way. Up probability denotes the probability that a node is up at a given time; it is also the fraction of time that a node is up in the long run. Our simulations are discrete-event simulations and the up probabilities determine the time intervals during which the nodes are up or down. Even though we use homogeneous up probabilities, each node behaves individually, but the long-run fraction of time a node is up is the same for all nodes.

All of the experimental results we report here use homogeneous up probabilities. In the experimental results we report here, 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. Due to lack of space, we do not show these results here.

Figure 2 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 [25], [26]. Each figure has a curve for $K = 1$ (Basic Replication Algorithm) and $K = 5$ (Top-K Replication Algorithm with $K = 5$). 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. In this case, each node is independent, i.e., there is no community at all. For the non-cooperative policy, each node again uses LRU cache replacement and uses the same amount of local storage as they would contribute to the community. 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 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 3 shows the fraction of misses for which the

[3]Note that we are considering large files, such as DVD videos, and a node with even 100 GB of storage would not be able to store many such videos.

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 4 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 Basic Replication 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 [25], [26]. Again, we show all the four combinations of parameters. The difference in how the theoretical optimal and the adaptive algorithm replicate files is striking. The optimal scheme replicates the more popular files much more aggressively than does the adaptive algorithm. Furthermore, the optimal scheme does not store the less popular files, whereas the adaptive algorithm provides temporary caching to the less popular files.

C. Top-K Most Frequently Requested Replication Algorithm

The Top-K Replication 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 replacement policy 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) replication 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 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 [25], [26], objects with different sizes should be ordered according to $\lambda_j(i)/b_j$, where b_j is the size of object j.

When node i receives a request (from any other node) for file j, it updates $\lambda_j(i)$ and checks 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 the requesting node X 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 considered a request, and the nodes could update their request rates and manage their storage with MFR accordingly. But it turns out that this approach does not give better performance than Top-K Replication Algorithm.

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:

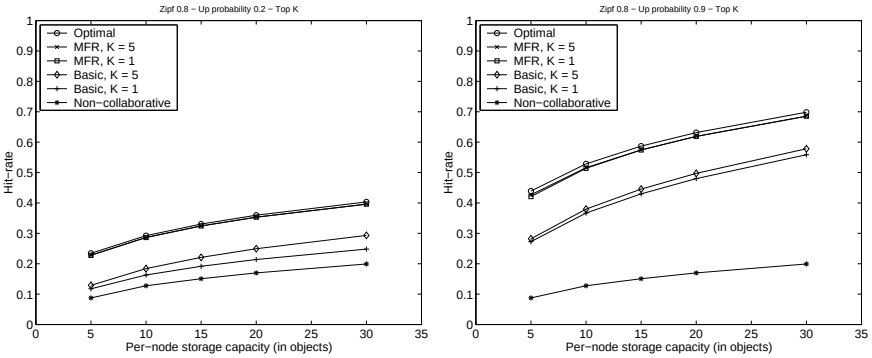
1) X uses overlay to determine i, the kth place winner for j.
2) X requests j from i.
   - Node i updates $\lambda_j(i)$.
   - If node i already has j, node i sends j to X; stop.
   - If node i does not have j but it should (according to MFR), i gets j, stores j and evicts files if necessary. Node i 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 locating the object (or determining that it is not available). 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 fraction of the total delay. In addition, downloads within the community typically 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.

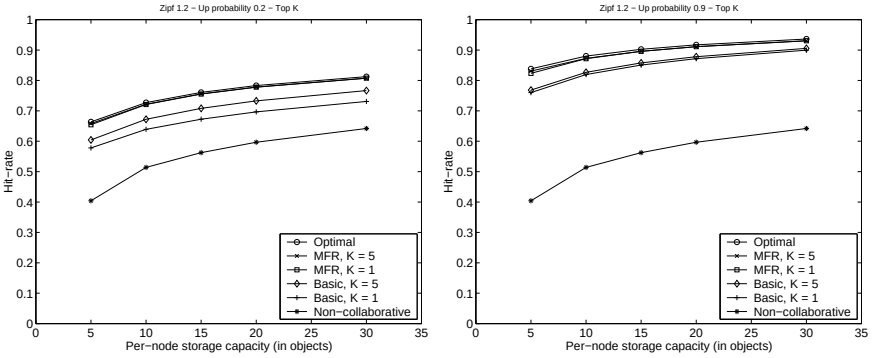
Figure 5 shows, as a function of file popularity, the number of replicas per file for the theoretical optimal and for Top-

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

[5]Node i could retrieve j from the outside, or it could ping the remaining internal winners for the file. (Such pings do not count as requests for j.)



(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.

Fig. 2. Hit probability as function of node storage capacity

K MFR Algorithm with $K = 5$. We see that, in contrast with Basic and Top-K Replication Algorithms, 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 2 compares the hit rate of MFR (with $K = 1$ and $K = 5$) with the Basic and Top-K Replication Algorithms and with the optimal hit rate. We see 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 differences between MFR and optimal replication/replacement are due to imperfect load-balancing in the DHT overlay 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.*

D. Performance Analysis of MFR

We now present an efficient performance evaluation technique for the MFR algorithm, which is not only accurate and efficient, but also sheds insight into the subtleties of the MFR algorithm.

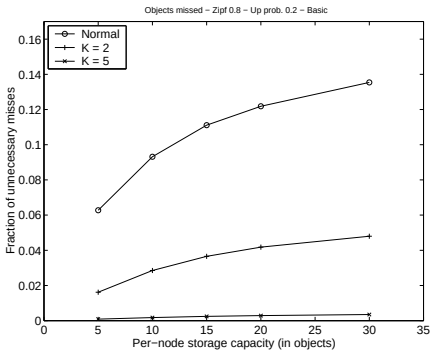
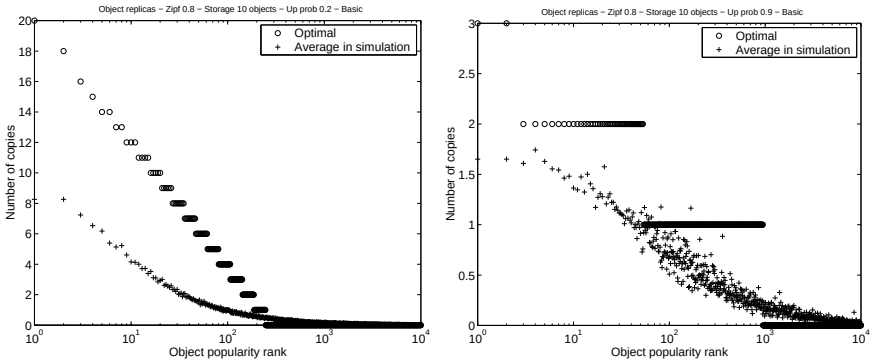


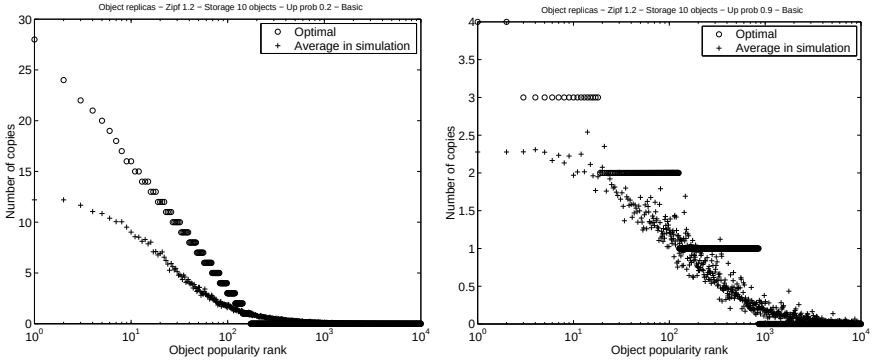
Fig. 3. Fraction of misses for Zipf = 1.2

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,

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 the ith node. Let J denote the number of distinct files, and let b_j denote the size 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



(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.

Fig. 4. Number of replicas per file with 10 files of per-node storage capacity, Basic Replication Algorithm with LRU replacement policy

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,

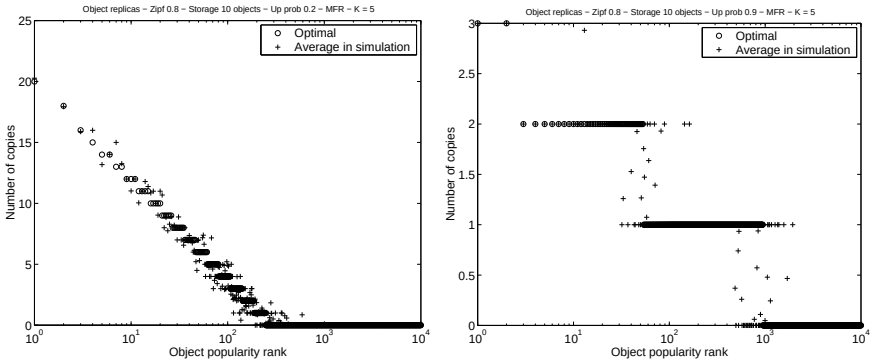
1) Find the file j that has the largest value of γ_j.
2) Sequentially examine the winning nodes for j until a node is found such that $T_i \geq b_j$ and $x_{ij} = 0$. Then,
   - Set $x_{ij} = 1$
   - Set $\gamma_j = \gamma_j(1 - p_i)$
   - Set $T_i = T_i - b_j$.

   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.

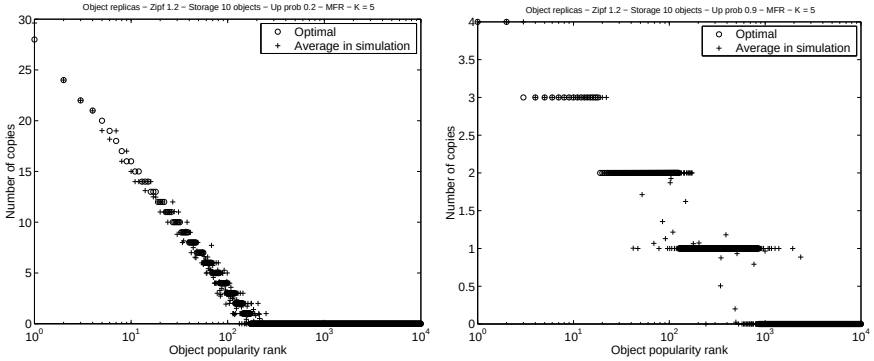
The replication profile provided by this procedure is typically very close to the steady-state profile obtained by the Top-I MFR algorithm. The two profiles may differ slightly due to how files of different sizes are replaced and packed in the nodes, and due to ties in Step 1. However, under the conditions 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 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 γ_j when 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.



(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.

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

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 node up probability, Zipf parameter, and node storage capacity. Each case had 100 nodes. For the same cases, we obtained the theoretical optimal solution (see [25], [26]). For all of the cases, the 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 I. 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.

TABLE I
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

Choosing the Correct K for Top-K Algorithms

Pinging a large number of nodes (i.e., large K) is likely to go unnoticed by the user because the download time of the actual object is likely several orders of magnitude longer than the time it takes to ping all K nodes. As our analysis and evaluation in this section shows, as K tends to I (the number of nodes), the performance of the MFR algorithm tends to the optimal performance, in most cases. Also, as shown in Figure 2, the Top-K Replication algorithm also improves its performance as K increases.

However, increasing the value of K will increase the

number of messages in the community as the nodes are searching for the files. Hence, it may be desirable to limit the number of nodes that are pinged by the different Top-K algorithms. In a real world community, the value of K could be determined adaptively during run-time and it could be different for different objects and nodes, but a small value would likely be sufficient.

The optimal value of K, i.e., tradeoff between higher hit-rate and smaller number of messages, depends on the individual node up/down dynamics, request rates, and object popularities. In a real world community, the value of K could be determined adaptively during run-time and it could be different for different objects and nodes, but a small value would likely be sufficient. As our results show, even the basic MFR algorithm ($K = 1$) has close to optimal performance and higher values of K are useful only when node up probabilities and shared storage per node are both small (see Figure 2(b), right hand graph).

V. 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 present two algorithms for solving the hot-spot problem and evaluate their performance.

One approach to this problem is to segment 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. We refer to this approach as the *fragmentation approach*. 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 [25], [26] we provide an upper bound on the performance of adaptive schemes that use erasures.

Another approach is to leave the files intact 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*. Naturally, it is also possible to combine the two approaches, such that files are fragmented and nodes have the possibility of refusing some requests. Our evaluation below will cover all three solutions: fragmentation, overflow, and fragmentation and overflow together.

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.

A. Fragmentation Approach

As mentioned above, in the fragmentation approach, we break each file into several fragments or chunks which are given unique names and stored individually in the DHT. In order to retrieve the file, a peer must retrieve all the fragments. We do not consider erasures in our evaluation, that is, we simply fragment a file into K chunks and each chunk must be retrieved (either from inside or outside the community).

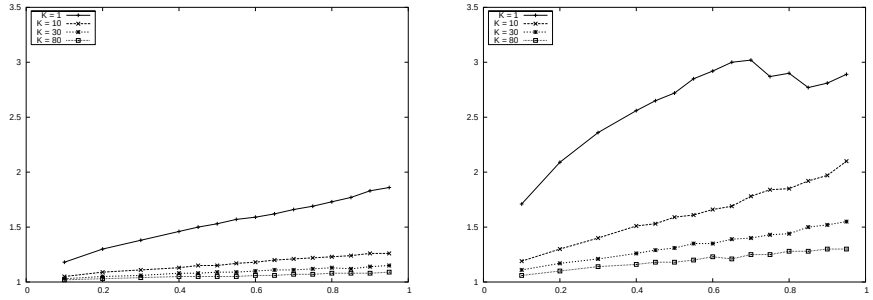
Because the fragments of a popular file are likely to stored on different peers (different first-, second-, etc. place nodes) and the fragments are much smaller than the original file, the file transfer load on the nodes becomes more balanced. The fragmentation approach is therefore a universal mechanism for balancing the file transfer load on the peers and it is independent of the actions individual peers (compare with the overflow approach below).

The possible downside of the fragmentation approach is that many fragments need to be available to maintain a high hit-rate. In other words, whereas in a system storing complete files, only 1 hit is needed to get the whole file from inside the community, in the fragmentation approach we need to get a hit for every single fragment to get the same effect as in the complete-file system. However, a miss in the fragmentation approach concerns only a single fragment which is much smaller than a file.

Our evaluation of the fragmentation approach covers the above two questions: How well is the load balanced between the peers and how much does the hit-rate decrease? We evaluated the approach using the same method as in Section IV, and report the results for the similar parameter combinations. In the results below, we have 100 peers, 10000 objects, two Zipf parameters (0.8 and 1.2), and different up probabilities for the peers. We also cover a different number of fragments from 1 up to 80 fragments per file. A typical fragment size in systems which use them is 256 KB, so in reality, 10–20 fragments can represent a music file, whereas 80 fragments is a small video. (Larger videos would have significantly more fragments, but as our results show, 80 fragments is sufficient to give a well-balanced load.)

Figure 6 shows the 90-percentile of the per-peer load as a function of the up probability for the two Zipf parameters. The y-axis represents the load relative to the "fair share" of each peer. We calculated the fair share by assuming that each peer would handle the same amount of traffic and compared the actual loads to this value. Hence, a value of 2 indicates that 90% of the peers in the experiment had to serve more than twice the number of files than their fair share was.

As a first observation, we can see that for Zipf 1.2 and 1 fragment (i.e., experiments in Section IV), the load is severely unbalanced. However, as we increase the number of fragments, the load balances itself quite well, and even for Zipf 1.2 remains below 1.3. Even though 1.3 means that the highly loaded peers are handling more than 30% over their fair share, this can be remedied with the overflow approach below.



(a) Zipf 0.8 (b) Zipf 1.2

Fig. 6. 90-percentile of per-peer load for fragmentation approach

Another interesting observation for Zipf 1.2 and 1 fragment is the behavior of the load curve with higher up probabilities, where the load no longer increases monotonically, but varies and can even decrease. Comparing figures 6(a) and 6(b), we see that with Zipf 0.8, all the load curves increase monotonically with the up probability. This is intuitive, since the more peers are up in general, the more of the load they have to handle (especially the first-place winners for popular objects). The load for Zipf 1.2 peaks earlier, at an up probability of around 0.65–0.7. The results in Figure 6 are calculated after several long experiment runs and we observed the same behavior for all larger Zipf-parameter values (values greater than 1.0, for lack of space these results are not reported here).

We believe that the explanation for this is as follows. With higher Zipf-parameter values, the popular files receive far more requests than the less popular files when compared to cases with lower Zipf-parameter values. This implies that a peer, who is among the first *few* winners for a popular object, is going to get a large number of requests. Furthermore, up probabilities in the range of 0.5–0.8 imply that peers are typically up, but not enough to make the first-place winner get almost all of the requests. In other words, the load of the most popular objects is spread over several peers, which, in turn, raises the 90-percentile of the load. The load of the most heavily loaded peer typically increases monotonically with the up probability, as can be expected (not shown in the figure).

In summary, the fragmentation approach is very efficient in balancing the load among the peers. The more fragments there are, the better the load is balanced. For smaller files, the load will remain slightly unbalanced because of the smaller number of fragments. For large videos, the larger number of fragments guarantees a relatively well-balanced load.

B. Overflow Approach

In the overflow approach, individual peers can refuse to serve a request which causes the request to be sent to the next winner, or to the outside, as appropriate. Because peers are independent, an individual peer can use the overflow approach to lower its load by an arbitrary amount, hence guaranteeing that a peer does not have to serve more requests than it is capable. This guarantee allows us to state that the overflow approach (with or without the fragmentation approach) can deliver us *any* load distribution. However, this guarantee comes at a price.

When a peer refuses to serve a request, another peer has to serve it, or the file has to be retrieved from the outside. Therefore, when a peer is using the overflow approach, the load on other peers increases *and/or* the hit-rate goes down. We now evaluate the effects of the overflow approach according to these two metrics. In our evaluation, we will consider the overflow approach on its own as well as together with the fragmentation approach.

Pure Overflow Approach

In the pure overflow approach, we consider complete files, i.e., files with only 1 fragment. We performed the same experiments as in Section V-A and measured how much the load increases on each peer. We then calculated the average of the *increases* in load and report it in the figures. Our evaluation of the increase in load does not take into account the limited storage of nodes, but allows each node to store as many objects needed. Although this is unrealistic, it gives us the upper bound for the load increase.

In our experiments, we determined the peers refusing to serve requests as follows. We randomly determined a fraction of the peers and each of these peers would then refuse a given percentage of the requests it receives. We varied the two percentages and report the values for 10 and 50% of the peers refusing either 10 or 90% of the requests. In our experiments in Section V-A, we had observed that in most cases, most of the peers (typically 75–90%) are handling more or less the same (“fair”) load, therefore in most cases, only a small fraction of peers would be refusing requests (assuming the goal of the refusals was to balance the load). This is covered by the 10% of the peers refusing either 10% or 50% of the requests[6]. The other parameter combinations give us insight into how the refusals affect the load and hit-rates when a large number of peers is refusing a large number of requests.

Figure 7 shows the increased load per peer as a function of the up probability for the 4 combinations and the 2 Zipf-parameters. The y-axis reports the *additional* load as a percentage of the old load that peers have to handle on average as a direct result of the refusals. As we can see, when a small number of peers is refusing a small number of requests, the

[6]These are the peers handling more load than shown in Figure 6

Peers	Storage	Overflow-MFR
10% peers, 10% traffic	10	0.72
10% peers, 10% traffic	30	0.8
50% peers, 90% traffic	10	0.66
50% peers, 90% traffic	30	0.75

TABLE II
EFFECT OF REFUSALS ON HIT-RATE.

overall effect on the load is negligible, on the order of 1%. This applies also to the case of 50% of the peers refusing 10% of the requests. In the worst case, half of the peers refusing 90% of the requests, the additional load on the other peers is about 5–6%. As can be expected, the overall increase in load is determined by the total amount of traffic being refused, i.e., how many peers are refusing times the amount of traffic they refuse.

In summary, the overflow approach can be used to reduce load, but for complete files this can imply an increase of load by 6% when nodes have high up probabilities.

Overflow and Fragmentation

We now turn to evaluating the overflow approach in the case where files are fragmented. We use the same parameters as in the above evaluation and Figure 8 shows the results for the case of 30 fragments per file. As the first observation, we can clearly see that the additional load is less than 0.5% in all the cases. This is because the fragments are relatively small (compared to the case of complete files shown in Figure 7) and the additional load from the fragments is better distributed over the peers, since all fragments are individually stored on the peers.

A further observation shows that the increase in load behaves similarly to the non-fragmented case.

Effect of Overflow on Hit-Rate

As mentioned above, the overflow approach increases load on other peers, but can also decrease the overall hit-rate of the community, if large enough number of peers is refusing a large amount of traffic.

We evaluated the decrease in hit-rate in the above cases using the same storage sizes as in Section IV. Table II shows the results for the case of 10% peers refusing 10% of the traffic and 50% of the peers refusing 90% of the traffic, for Zipf parameter 1.2 and per-node storage of 10 and 30 objects. Other experiments yielded similar results and are not shown here. In the table, the column *Peers* defines the amount of refused traffic and column *Storage* indicates the per-node storage in objects. Column *Overflow-MFR* shows the reduced hit-rate after the refused traffic. For reference, the hit-rate obtained by the MFR algorithm is 0.73 for the case of 10 objects and 0.81 for 30 objects. For the non-cooperative strategy the hit-rates for 10 and 30 objects are 0.52 and 0.63, respectively.

We observed that the refusals to serve traffic have the same effect when we would reduce the amount of storage in the nodes. In other words, if 50% nodes are refusing 90% of the traffic, the community has roughly the same hit-rate as a community which serves all the traffic, but has only 55% of the storage of the community refusing traffic. The explanation behind this is intuitive. If a peer refuses to serve a request, the request gets passed to the next-place winner for the object, who will then see a higher request rate for that object. This increases the priority of that object in the MFR algorithm and might kick a less requested object out of the storage. This effect propagates along all the nodes and causes the least requested objects to be completely evicted from the community, thus effectively reducing the amount of storage provided by the community.

C. Summary

From our evaluation of the hot spot algorithms, we can make the following conclusions:

- Without any load balancing algorithms, the load is heavily unbalanced.
- The fragmentation approach is very successful in balancing the load between the nodes. Even a moderate number of fragments gives us a well-balanced load and the balance improves as the number of fragments increases.
- The pure overflow approach allows a peer to reduce its load to any desired level at the cost of about 5% increase in load for other peers.
- The overflow algorithm with fragmentation gets the benefits of the fragmentation approach for well-balanced load and the ability of the overflow approach for individual nodes to reduce their load. In contrast to the pure overflow approach, the overflow with fragmentation approach increases load only by about 0.5%.
- The amount of refused traffic effectively reduces the amount of storage provided by the nodes, thus reducing the hit-rate.

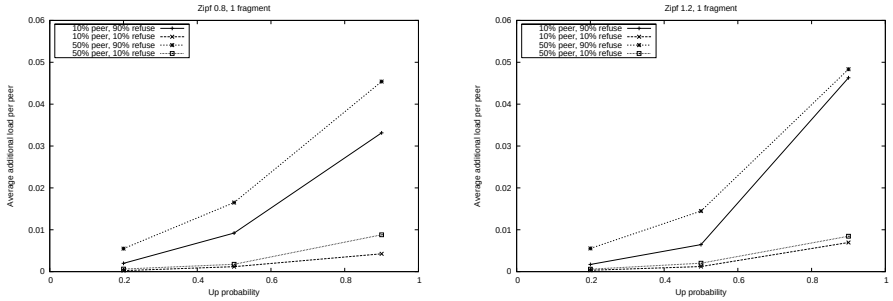
We can therefore state that using the overflow with fragmentation approach will eliminate differences in the load each node sees.

VI. CONCLUSIONS

P2P file sharing 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.

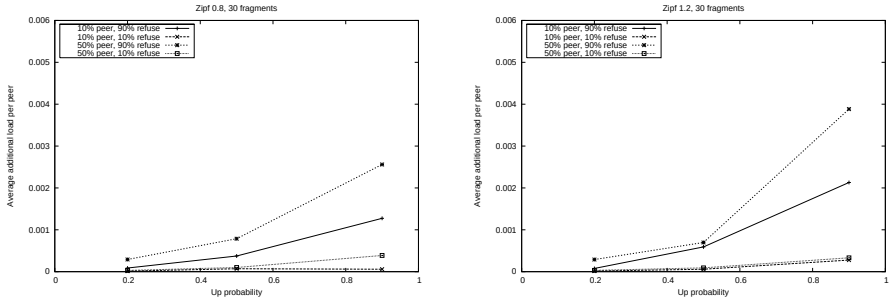
One of the features of structured, DHT-based P2P file sharing is that the application has significant control on where and how many replicas are generated. The contribution of this paper is threefold.

- First, 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. The methodology applies to designs that use erasures.
- Third, we have evaluated the load balancing properties of our replication algorithms and found that the basic



(a) Zipf = 0.8 (b) Zipf = 1.2

Fig. 7. Increased load due to overflow approach and complete files



(a) Zipf = 0.8 (b) Zipf = 1.2

Fig. 8. Increased load due to overflow approach and 30 fragments per file

algorithms result in a highly unbalanced load. We have proposed two load balancing algorithms, fragmentation and overflow approach, and have shown that they are very effective at balancing the load among the nodes.

REFERENCES

[1] eDonkey2000, http://www.edonkey2000.com
[2] Gnutella, http://www.gnutella.com
[3] Cachelogic, http://www.cachelogic.com
[4] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, Wide-area cooperative storage with CFS, *ACM SOSP 2001*, October 2001
[5] P. Druschel and A. Rowstron, "PAST: A large-scale, persistent peer-to-peer storage utility", *HotOS VIII*, , May 2001
[6] J. Kubiatowicz et al. "OceanStore: An Architecture for Global-Scale Persistent Storage," *ASPLOS 2000*, Nov. 2000.
[7] I. Stoica, D. Adkins, S. Zhaung, S. Shenker, and S. Surana, "Internet Indirection Infrastructure," *Proceedings of ACM SIGCOMM'02*, Pittsburgh, August 2002.
[8] 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)*, 20 (8), October 2002.
[9] S. Iyer, A. Rowstron and P. Druschel, "SQUIRREL: A Decentralized, Peer-to-Peer Web Cache", *PODC 2002*.
[10] 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
[11] H. Weatherspoon and J. D. Kubiatowicz, "Erasure Coding vs. Replication: A Quantitative Comparison," *IPTPS '02*.
[12] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *SIGCOMM*, 2001
[13] 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
[14] 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.
[15] E. Cohen and S. Shenker, "Replication strategies in unstructured peer-to-peer networks," in *SIGCOMM 2002*.
[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," In *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.
[25] J. Kangasharju, K.W. Ross, D.A. Turner, "Optimal Content Replication in Peer-to-Peer Communities", (work in progress).
[26] J. Kangasharju, "Internet Content Distribution", PhD Thesis, University of Nice Sophia Antipolis/Eurecom, April 2002.