# Modeling and Caching of Peer-to-Peer Traffic

Osama Saleh
School of Computing Science
Simon Fraser University
Surrey, BC, Canada

Mohamed Hefeeda
School of Computing Science
Simon Fraser University
Surrey, BC, Canada

***Abstract*— Peer-to-peer (P2P) file sharing systems generate a major portion of the Internet traffic, and this portion is expected to increase in the future. We explore the potential of deploying proxy caches in different Autonomous Systems (ASes) with the goal of reducing the cost incurred by Internet service providers and alleviating the load on the Internet backbone. We conduct a measurement study to model the popularity of P2P objects in different ASes. Our study shows that the popularity of P2P objects can be modeled by a Mandelbrot-Zipf distribution, regardless of the AS. Guided by our findings, we develop a novel caching algorithm for P2P traffic that is based on object segmentation, and partial admission and eviction of objects. Our trace-based simulations show that with a relatively small cache size, less than 10% of the total traffic, a byte hit rate of up to 35% can be achieved by our algorithm, which is close to the byte hit rate achieved by an off-line optimal algorithm with complete knowledge of future requests. Our results also show that our algorithm achieves a byte hit rate that is at least 40% more, and at most triple, the byte hit rate of the common web caching algorithms. Furthermore, our algorithm is robust in face of aborted downloads, which is a common case in P2P systems.**

## I. Introduction

Peer-to-peer (P2P) file-sharing systems have gained tremendous popularity in the past few years. More users are continually joining such systems and more objects are being made available, enticing even more users to join. Currently, traffic generated by P2P systems accounts for a major fraction of the Internet traffic [1], and it is expected to increase [2]. The sheer volume and expected high growth of P2P traffic have negative consequences, including: (i) significantly increased load on the Internet backbone, hence, higher chances of congestion; and (ii) increased cost on Internet Service Providers (ISPs) [3], hence, higher service charges for all Internet users. A potential solution for alleviating those negative impacts is to *cache* a fraction of the P2P traffic such that future requests for the same objects could be served from a cache in the requestor's autonomous system (AS).

Caching in the Internet has mainly been considered for web and video streaming traffic, with little attention to the P2P traffic. Many caching algorithms for web traffic [4] and for video streaming systems [5] have been proposed and analyzed. Directly applying such algorithms to cache P2P traffic may not yield the best cache performance, because of the different traffic characteristics and caching objectives. For instance, reducing user-perceived access latency is a key objective for web caches. Consequently, web caching algorithms often incorporate information about the cost (latency) of a cache miss when deciding which object to cache/evict. Although latency is important to P2P users, the goal of a P2P cache is often focused on the ISP's primary concern; namely, the amount of bandwidth consumed by large P2P transfers. Consequently, byte hit rate, i.e., minimizing the number of bytes transfered, is more important than latency. Moreover, P2P objects tend to be larger than web objects [1] reducing the number of complete objects that can be held in a cache.

Furthermore, although objects in P2P and video streaming systems share some characteristics, e.g., immutability and large size, streaming systems impose stringent timing requirements. These requirements limit the flexibility of caching algorithms in choosing which segments to store in the cache. Therefore, new caching algorithms that consider the new traffic characteristics and system objectives need to be designed and evaluated.

In this paper, we first develop a deeper understanding of the P2P traffic characteristics that are *relevant to caching*, such as object popularity. We do that via a three-month measurement study on a popular file-sharing system. Then, we design and evaluate a novel P2P caching algorithm for object admission, segmentation and replacement.

Specifically, our contributions can be summarized as follows. First, we develop new models for P2P traffic based on our measurement study. We show that the popularity of P2P objects can be modeled by a Mandelbrot-Zipf distribution, which is a generalized form of Zipf-like distributions with an extra parameter. This extra parameter captures the flattened head nature of the popularity distribution observed near the lowest ranked objects in our traces. The flattened head nature has also been observed by a previous study [1], but no specific distribution was given. Second, we analyze the impact of the Mandelbrot-Zipf popularity model on caching and show that relying on object popularity alone may not yield high hit rates/byte hit rates. Third, we design a new caching algorithm for P2P traffic that is based on segmentation, partial admission and eviction of objects.

We perform trace-based simulations to evaluate the performance of our algorithm and compare it against common web caching algorithms, such as LRU, LFU and GDS [6], and a recent caching algorithm proposed for P2P systems [7]. Our results show that with a relatively small cache size, less than 10% of the total traffic, a byte hit rate of up to 35% can be achieved by our algorithm, which is close to the byte hit rate achieved by an off-line optimal algorithm with complete

1-4244-0593-9/06/$20.00 ©2006 IEEE

knowledge of future requests. Our results also show that our algorithm achieves a byte hit rate that is at least 40% more, and at most triple, the byte hit rate of the common web caching algorithms.

The rest of this paper is organized as follows. In Section II, we summarize the related work. Section III describes our measurement study, presents a new model for object popularity, and analyzes the effect of this model on cache performance. Our P2P caching algorithm is described in Section IV. We evaluate the performance of our algorithm using trace-based simulation in Section V. Section VI concludes the paper.

## II. Related Work

We first summarize previous P2P measurement studies, justifying the need for a new study. Then, we contrast our caching algorithm with other P2P, web and multimedia caching algorithms.

Several measurement studies have analyzed various aspects of P2P systems. Gummadi et al. [1] study the object characteristics of P2P traffic in Kazaa and show that P2P objects are mainly immutable, multimedia, large objects that are downloaded at most once. The study demonstrates that the popularity of P2P objects does not follow Zipf distribution, which is usually used to model the popularity of web objects [8]. The study provides a simulation method for generating P2P traffic that mimics the observed popularity curve, but it does not provide any closed-form models for it. Sen and Wang [9] study the aggregate properties of P2P traffic in a large-scale ISP, and confirm that P2P traffic does not obey Zipf distribution. Their observations also show that few clients are responsible for most of the traffic. Klemm et al. [10] use two Zipf-like distributions to model query popularity in Gnutella. Because the authors are mainly interested in query popularity, they do not measure object popularity as defined by actual object transfers.

While these measurement studies provide useful insights on P2P systems, they were not explicitly designed to study caching P2P traffic. Therefore, they did not focus on analyzing the impact of P2P traffic characteristics on caching. The study in [1] highlighted the potential of caching and briefly studied the impact of traffic characteristics on caching. But the study was performed in only one network domain.

The importance and feasibility of caching P2P traffic have been shown in [11] and [3]. The study in [11] indicates that P2P traffic is highly repetitive and responds well to caching. Whereas the authors of [3] show that current P2P protocols are not ISP-friendly, because they impose unnecessary traffic on ISPs. The authors suggest deploying caches or making P2P protocols locality-aware. Both [11] and [3] do not provide any algorithm for caching.

The closest work to ours is [7], where two cache replacement policies for P2P traffic are proposed. These two policies are: MINRS (Minimum Relative Size), which evicts the object with the least cached fraction, and LSB (Least Sent Byte), which evicts the object which has served the least number of bytes from the cache. Our simulation results show that our algorithm outperforms LSB, which is better than MINRS according to the results in [7].

Partial and popularity-based caching schemes for web caching, e.g., [12], and video streaming, e.g., [13], [14] have been proposed before. [12] proposes a popularity-aware greedy-dual size algorithm for caching web traffic. Because the algorithm focuses on web objects, it does not consider partial caching, which is critical for P2P caching due to large sizes of objects. Jin et al. [13] consider partial caching based on object popularity, encoding bit rate, and available bandwidth between clients and servers. Their objectives are to minimize average start-up delays and to enhance stream quality. In contrast, our partial caching approach is based on the number of bytes served from each object normalized by its cached size. This achieves our objective of maximizing the byte hit rate without paying much attention to latency. A partial caching algorithm for video-on-demand systems is proposed in [14], where the cached fraction of a stream is proportional to the number of bytes played back by all clients from that stream in a time slot. Unlike our algorithm, the algorithm in [14] periodically updates the fractions of *all* cached streams, which adds significant overhead on the cache.

Finally, our caching algorithm is designed for P2P systems, which contain multiple workloads corresponding to various types of objects. This is in contrast to the previous web and streaming caching algorithms which are typically optimized for only one workload.

## III. Modeling P2P Traffic

We are interested in deploying caches in different autonomous systems (ASes) to reduce the WAN traffic imposed by P2P systems. Thus, our measurement study focuses on measuring the characteristics of P2P traffic that would be observed by these *individual* caches, and would impact their performance. Such characteristics include object popularity, and number of P2P clients per AS. We measure such characteristics in several ASes of various sizes. In this section, we describe our measurement methodology and present our findings.

### A. Measurement Methodology

We conduct a *passive* measurement study of the Gnutella file-sharing network [15]. For the purposes of our measurement, we modify a popular Gnutella client called Limewire [16]. We choose to conduct our measurement on Gnutella because (i) it supports the super-peer architecture which facilitates non-intrusive passive measurements by observing traffic passing through super peers; and (ii) it is easier to modify since it is an open source protocol.

Previous studies show that Gnutella is similar to other P2P systems. For example, early studies on the fully-distributed Gnutella and the index-based Napster systems found that clients and objects in both systems exhibit very similar characteristics such as the number of files shared, session duration, availability and host uptime [17]. Another study on BitTorrent [18] made similar observations regarding object

characteristics and host uptime. Also the non-Zipf behavior of object popularity in Gnutella (as we show later) has been observed before in Kazaa [1]. Therefore, we believe that the Gnutella traffic collected and analyzed in our study is representative of P2P traffic in general.

According to the Gnutella protocol specifications peers exchange several types of messages including PING, PONG, QUERY and QUERYHIT. A QUERY message contains search keywords, a TTL field and the address of the immediate neighbor which forwarded the message to the current peer. Query messages are propagated to all neighbors in the overlay for a hop distance specified by the TTL field. A typical value for TTL is seven hops. If a peer has one or more of the requested files, it replies with a QUERYHIT message. A QUERYHIT message is routed on the reverse path of the QUERY message it is responding to, and it contains the name and the URN (uniform resource name) of the file, the IP address of the responding peer, and file size. Upon receiving replies from several peers, the querying peer chooses a set of peers and establishes direct connections with them to retrieve the requested file.

Limewire has two kinds of peers: *ultra-peers*, characterized by high bandwidth and long connection periods, and *leaf-peers* which are ordinary peers that only connect to ultra-peers. We run our measurement node in an ultra-peer mode and allow it to maintain up to 500 simultaneous connections to other Gnutella peers. On average, our measurement node was connected to 279 nodes, 63% of which were ultra peers. Our measurement node passively recorded the contents of all QUERY and QUERYHIT messages passing through it without injecting any traffic into the network.

The measurement study was conducted between 16 January 2006 and 16 April 2006. Our measurement peer was located at Simon Fraser University, Canada. But since the Gnutella protocol does not favor nodes based on their geographic locations [10], we were able to observe peers from many ASes across the globe. During the three months of the measurement, we recorded more than 214 million QUERY messages and 107 million QUERYHIT messages issued from more than 20 million peers distributed over more than 16 thousand different ASes. Table I shows the number of objects and hosts discovered in several ASes. The large scale of our measurement study enables us to draw solid conclusions about P2P traffic. The measurement data is stored in several trace files with a total size of approximately 20 giga bytes. The trace files are available to the research community at [19].

### *B. Measuring and Modeling Object Popularity*

In this subsection, we explain how we measure object popularity in different ASes. Then, we present and validate a simple, and fairly accurate, popularity model for objects in P2P systems.

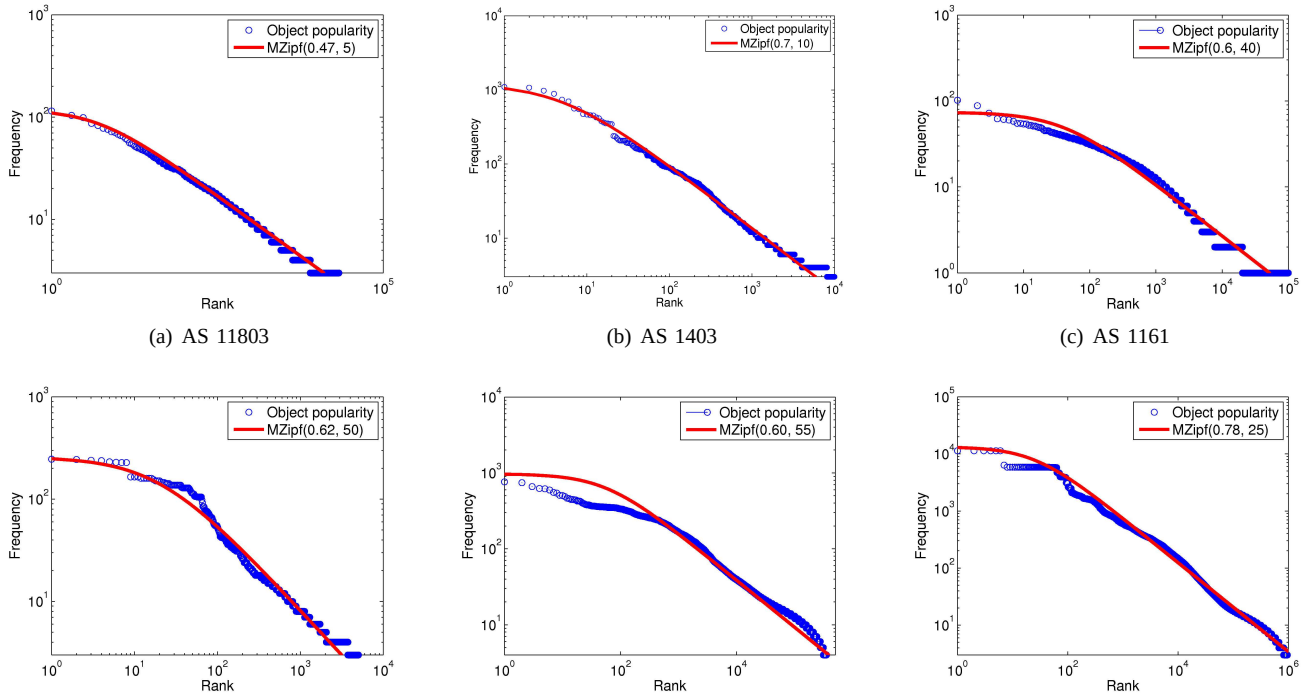
The relative popularity of an object is defined as the probability of requesting that object relative to other objects. Object popularity is critical for the performance of the cache. Intuitively, storing the most popular objects in the cache is expected to yield higher hit rates than storing any other set of objects.

Since we are primarily interested in the performance of individual caches, we measure the popularity of objects in *each* AS. To measure the popularity of an object in a specific AS, we count the number of replicas of that object in the AS considered. Number of replicas indicates the number of downloads that were completed in the past. This means that if a cache were deployed, it would have seen a similar number of requests. This assumes that most of the downloads were supplied by peers from outside the AS, which is actually the case because peers in most current P2P networks have no sense of network proximity and thus do not favor local peers over non-local peers. In fact, previous studies [1] have shown that up to 86% of the requested P2P objects were downloaded from peers outside the local network even though they were locally available.

To count the number of replicas of a given object, we extract from our trace all QUERYHIT messages which contain the unique ID (URN) of that object. QUERYHIT messages contain the IP addresses of the responding nodes that have copies of the requested object. We can determine the number of replicas by counting the number of unique IP addresses. Then, we map these unique IP addresses to their corresponding ASes by using the GeoIP database [20].

We compute the popularity of each object in each of the top 16 ASes (in terms of sending and receiving messages). These top 16 ASes contribute around 38% of the total traffic seen by our study. We also compute the popularity across all ASes combined. We rank objects based on their popularity, and we plot popularity versus rank. Fig. 1 shows a sample of our results. Results for other ASes are similar and are given in the technical report [21]. As shown in the figure, there is a *flattened* head in the popularity curve of P2P objects. This flattened head indicates that objects at the lowest ranks are not as popular as Zipf-like distributions would predict. This flattened head phenomenon could be attributed to two main characteristics of objects in P2P systems: immutability and large sizes. The immutability of P2P objects eliminates the need for a user to download an object more than once. This download at most once behavior has also been observed in previous studies [1]. The large size of objects, and therefore the long time to download, may make users download only objects that they are really interested in. This is in contrast to web objects, which take much shorter times to download, and therefore, users may download web objects even if they are of marginal interest to them. These two characteristics reduce the total number of requests for popular objects.

Fig. 1 also shows that, unlike the case for web objects [8], using a Zipf-like distribution to model the popularity of P2P objects would result in a significant error. In log-log scale, the Zipf-like distribution appears as a straight line, which can reasonably fit most of the popularity distribution except the left-most part, i.e., the flattened head. A Zipf-like distribution would greatly overestimate the popularity of objects at the lowest ranks. These objects are the most important to caching




(a) AS 11803 (b) AS 1403 (c) AS 1161

(d) AS 95 (e) AS 14832 (f) All ASes

Fig. 1. Object popularity in P2P traffic can be modeled by Mandelbrot-Zipf Distribution.

mechanisms, because they are the good candidates to be stored in the cache.

We propose a new model that captures the flattened head of the popularity distribution of objects in P2P systems. Our model uses the Mandelbrot-Zipf distribution [22], which is the general form of Zipf-like distributions. The Mandelbrot-Zipf distribution defines the probability of accessing an object at rank $i$ out of $N$ available objects as:

$$p(i) = \frac{K}{(i+q)^{\alpha}}, \tag{1}$$

where $K = \sum_{i=1}^{N} 1/(i+q)^{\alpha}$, $\alpha$ is the skewness factor, and $q \geq 0$ is a parameter which we call the *plateau* factor. $q$ is so called because it is the reason behind the plateau shape near to the left-most part of the distribution. Notice that the higher the value of $q$, the more flattened the head of the distribution will be. When $q = 0$, Mandelbrot-Zipf distribution degenerates to a Zipf-like distribution with a skewness factor $\alpha$. Fig. 2 compares Zipf distribution versus Mandelbrot-Zipf distribution for different $q$ values. Notice that, there is about an order of magnitude difference in frequency between the two distributions at the lowest ranks.

To validate this popularity model, we fit the popularity distributions of objects in each of the top 16 ASes to Mandelbrot-Zipf distribution using the Matlab distribution fitting tool. Our results, some of them are shown in Fig. 1, indicate that Mandelbrot-Zipf distribution models the popularity of P2P objects reasonably well. Table I summarizes $\alpha$ and $q$ values for eight representative ASes. We observe that a typical value for

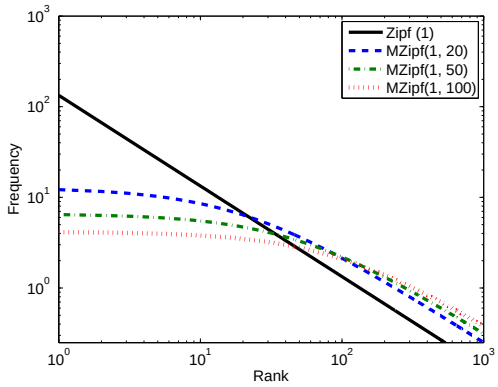
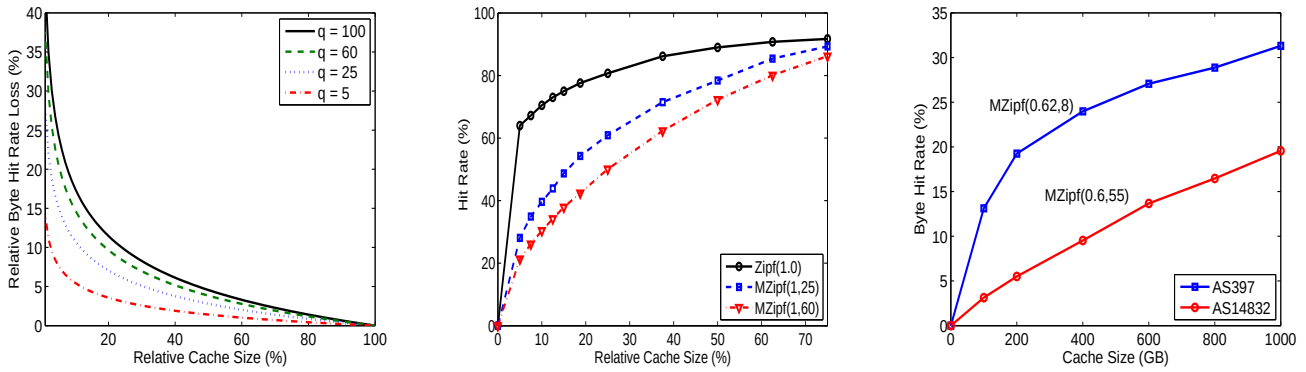



Fig. 2. Zipf versus Mandelbrot-Zipf for different $q$ (plateau factor) values.

$\alpha$ is between 0.4 and 0.70, and a typical value for $q$ is between 5 and 60. By inspecting our traces, we found that ASes with higher number of clients usually have smaller values for $q$. For example, in Table I, AS 9548 has 464,511 seen in our trace and it has a small $q$ value ($q$ = 5). On the other hand, AS 95 has much smaller number of clients, 6,121, and a higher value of $q$ (q = 50). This is intuitive because the number of clients is an upper bound on the number of times an object could be downloaded.

### C. The Effect of Mandelbrot-Zipf Popularity on Caching

In this subsection, we analyze the impact of the Mandelbrot-Zipf popularity model on the cache hit rate and byte hit rate using simple analysis and simulation.




(a) Hit rate loss under LFU (analytic). (b) Hit rate under LRU (simulation). (c) Byte hit rate under optimal (trace-based).

Fig. 3. Effect of Mandelbrot-Zipf popularity distribution on the cache performance.

TABLE I

OBJECT POPULARITY AND TRAFFIC CACHEABILITY IN SEVERAL AS DOMAINS.

| AS No. | number of unique IPs | number of unique objects | MZipf $(\alpha, q)$ | % of cache-able traffic |
|---|---|---|---|---|
| 9548 | 464, 551 | 570, 122 | (0.65, 5) | 42.30 |
| 95 | 6, 121 | 87, 730 | (0.6, 50) | 54.47 |
| 2609 | 13, 958 | 181, 184 | (0.55, 25) | 54.75 |
| 397 | 368, 604 | 418, 263 | (0.62, 8) | 48.72 |
| 1161 | 1000 | 135, 921 | (0.60, 40) | 16.67 |
| 1859 | 437, 901 | 767, 890 | (0.53, 10) | 46.16 |
| 14832 | 515 | 46, 704 | (0.60, 55) | 95.34 |
| 18538 | 337 | 33, 198 | (0.62, 60) | 93.74 |

We start with a simple analysis of an LFU (Least Frequently Used) policy. Under LFU, the $C$ most popular objects are stored in the cache. For simplicity, we assume that all objects have the same size and the skewness parameter $\alpha$ is 1. We are mainly interested in exploring the impact of the plateau factor $q$ on the hit rate. The hit rate $H$ of an LFU cache is given by:

$$H = \sum_{i=1}^{C} p(i) = \sum_{i=1}^{C} \frac{K}{(i+q)}$$
$$\approx \int_{i=1}^{C} \frac{K}{(i+q)} = K \ln\left(\frac{1+C/q}{1+1/q}\right). \quad (2)$$

Eq. (2) implies that increasing $q$ results in a decrease in hit rate. When $q \to \infty$, i.e., the head is very flat, the hit rate approaches zero. In contrast, for a Zipf-like popularity distribution ($q = 0$), the hit rate is $H = K \ln C$. To further illustrate the impact of Mandelbrot-Zipf on the cache performance, we plot in Fig. 3(a) the *relative loss* in hit rate between Zipf and Mandelbrot-Zipf distributions. The relative loss in hit rate is computed as $(H^{Zipf} - H^{MZipf})/H^{Zipf}$, where $H^{Zipf}$ and $H^{MZipf}$ are the hit rates achieved by an LFU cache if the popularity follows Zipf and Mandelbrot Zipf distributions, respectively. As the figure shows, significant loss in hit rate could be incurred because of the flattened-head nature of the Mandelbrot-Zipf popularity distribution. The loss in hit rate is higher for smaller relative cache sizes and larger values of $q$.

Next, we consider an LRU (Least Recently Used) cache. We use simulation to study the impact of the popularity model on the hit rate. We generate synthetic traces as follows. We consider $4,000$ equal-sized objects. We randomly generate requests for these objects according to the Zipf and Mandelbrot-Zipf distributions. We run the traces through an LRU cache with a relative cache size that varies between 0 and 100%. We compute the hit rate in each case. The results are shown in Fig. 3(b). As the figure indicates, the situation is even worse under LRU: higher drops in hit rates are observed.

Finally we use traces from our measurement study and compute the maximum achievable byte hit rate in two different ASes. We pick two ASes from our traces with similar $\alpha$ values but different $q$ values: AS 397 with $q = 8$ and AS 14832 with $q = 55$. We use an optimal off-line algorithm which looks at the trace of each AS and stores in the cache the objects which will serve the most number of bytes, hence, achieves the highest byte hit rate. We perform trace-based simulation and compute the byte hit rate under various cache sizes for both ASes. As can be seen from Fig. 3(c), with a cache size of 400 GB, a byte hit rate of 24% is achieved under AS 397, while only 9% byte hit rate is achieved under AS 14832 using the same cache size. This means that the top popular objects which can fit in a cache size of 400 GB receive 24% of the total outgoing requests in AS 397, in comparison to 9% of the total outgoing requests in AS 14832.

These observations and experiments imply that caching schemes that capitalize on object popularity *alone* may not yield high hit rates/byte hit rates and may not be very effective in reducing the ever-growing P2P traffic.

## IV. P2P CACHING ALGORITHM

With the understanding of the P2P traffic we developed, we design and evaluate a novel P2P caching scheme based on segmentation and partial caching. Partial caching is necessary because objects in P2P systems are large, ranging from 20 KB to 10 GB. Furthermore, P2P objects are composed of multiple workloads, each workload corresponds to a different content type (e.g., audio, video, documents), and occupies a subrange of the total object size range. A detailed discussion on the characteristics of P2P objects is given in the accompanying technical report [21]. Moreover, because of the Mandelbrot-Zipf popularity model, the most popular objects may not

receive too many requests. Thus storing an entire object upon a request may waste cache space. P2P caching should take a conservative approach towards admitting new objects into the cache to reduce the cost of storing unpopular objects. To achieve this objective, our algorithm divides objects into smaller segments and incrementally admits more segments of an object to the cache as the object receives more requests.

We use different segment sizes for different workloads, because using the same segment size for all workloads may favor some objects over others and introduce additional cache management overhead. For example, if we use a small segment size for all workloads, large objects will have large number of segments. This introduces a high overhead of managing large number of segments within each object. Such overhead includes locating which segments of an object are cached, and deciding which segments to evict. On the other hand, using a large segment size for all workloads will favor smaller objects since they will have smaller number of segments and get cached quicker. Making segment size relative to object size has the same two problems. This is because the range of P2P object sizes extends to several gigabytes. Thus using relative segmentation, as a percentage of an object size, may result in large objects having segments that are orders of magnitude larger than unsegmented smaller object.

Consistent with our observations of multiple workloads, we use four different segment sizes. For objects with sizes less than 10 MB, we use a segment size of 512 KB, for objects between 10 MB and 100 MB, a segment size of 1 MB, and for objects between 100 MB and 800 MB a segment size of 2 MB. Objects whose size exceeds 800 MB are divided into segments of 10 MB in size. We call this segmentation scheme variable segmentation scheme. Note that our segmentation procedure is inline with most segmentation schemes used by real P2P systems. Our analysis of more than 50,000 unique torrent files shows that the vast majority of BitTorrent objects are segmented into 256 KB, 512 KB, 1 MB and 2 MB segments. E-Donkey uses segments of size 9.28 MB [23]. Depending on the client implementation, segment size in Gnutella can either be a percentage of object size, as in BearShare [24], or fixed at few hundred KBs [15]. We opt not to follow protocol-specific segmentation scheme so as to make our algorithm independent of the underlying P2P protocol. In addition, protocol-specific segmentation scheme might not always be inline with the objective of maximizing byte hit rate. This is confirmed by our evaluation in Section V.

The basic idea of our algorithm is to cache of each object a portion that is proportional to its popularity. That way popular objects will be incrementally given more cache space than unpopular objects. To achieve this goal, our algorithm caches a constant portion of an object when it first sees that object. As the object receives more hits, its cached size increases. The rate at which the cached size increases grows with the number of hits an object receives. The unit of admission into the cache is one segment; that is the smallest granularity our algorithm caches/evicts is one segment.

For each workload, the cache keeps the average object size in that workload. Denote the average object size in workload $w$ as $\mu_w$. Further let $\gamma_i$ be the number of bytes served from object $i$ normalized by its cached size. That is, $\gamma_i$ can be considered as number of times each byte in this object has been served from the cache. The cache ranks objects according to their $\gamma$ value, such that for objects ranked from 1 to $n$, $\gamma_1 \geq \gamma_2 \geq \gamma_3 \geq \cdots \geq \gamma_n$. We refer to an object at rank $i$ simply as object $i$. When an object is seen for the first time, only one segment of it is stored in the cache. If a request arrives for an object of which at least one segment is cached, the cache computes the number of segments to be added to this object's segments as $(\gamma_i/\gamma_1)\mu_w$, where $\mu_w$ is the mean of the object size in workload $w$ which object $i$ belongs to. Notice that this is only the number of segments the cache could store of object $i$. But since downloads can be aborted at any point during the session, the number of segments *actually* cached upon a request, denoted by $k$, is given by

$$k = \min[missed,\ \max(1, \frac{\gamma_i}{\gamma_1}\mu_w)]\ , \tag{3}$$

where $missed$ is the number of requested segments not in the cache. This means that the cache will stop storing uncached segments if the client fails or aborts the download, and that the cache stores at least one segment of an object.

The pseudo-code of our P2P caching algorithm appears in Fig. 4. At a high level, the algorithm works as follows. The cache intercepts client requests and extracts the object ID and the requested range. If no segments of the requested range are in the cache, the cache stores at least one segment of the requested range. If the entire requested range is cached, the cache will serve it to the client. If the requested range is partially available in the cache, the cache serves the cached segments to the client, and decides how many of the missing segments to be cached using Eq. (3). In all cases, the cache updates the average object size of the workload to which the object belongs and the $\gamma$ values of the requested object.

**request(object $i$, requested range)**

1. **if** object $i$ is not in the cache
2. add one segment of $i$ to cache, evicting if necessary
3. **else**
4. hit = cached range $\cap$ requested range
5. $\gamma_i$ += hit / cached size of $i$
6. missed = (requested range - hit)/segment size
7. $k = \min[\text{missed}\ ,\ \max(1, \frac{\gamma_i}{\gamma_1}\mu_w)]$
8. **if** cache does not have space for $k$ segments
9. evict $k$ segments from the least valued object(s)
10. add $k$ segments of object $i$ to the cache
11. **return**

Fig. 4. P2P caching algorithm.

The algorithm uses a priority queue data structure to store objects according to their $\gamma_i$ values. When performing eviction, segments are deleted from the least valued objects. Currently,

our algorithm evicts contiguous segments without favoring any segments over others. This is because P2P downloads are likely to start from anywhere in the file [7]. Our algorithm needs to perform $O(\log N)$ comparisons with every hit or miss, where $N$ is the number of objects in the cache. But since objects are large, this is a reasonable cost considering the small number of objects the cache will contain.

## V. Evaluation

In this section, we use trace-driven simulation to study the performance of our P2P caching algorithm, and compare it against three common web caching algorithms (LRU, LFU and GDS) and a recent caching algorithm proposed for P2P systems.

### A. Experimental Setup

We use traces obtained from our measurement study to conduct our experiments. Our objective is to study the effectiveness of deploying caches in several ASes. Thus, we collect information about objects found in a certain AS and measure the byte hit rate that could be achieved if a cache were to be deployed in that AS. We use the byte hit rate as the performance metric because we are mainly interested in reducing the WAN traffic.

We run several AS traces through the cache and compare the byte hit rate achieved using several caching algorithms. In addition to our algorithm, we implement the Least Recently Used (LRU), Least Frequently Used (LFU), Greedy-Dual Size (GDS) [6] and Least Sent Bytes (LSB) [7] algorithms. We also implement the off-line optimal (OPT) algorithm and use it as a benchmark for the performance of other algorithms. LRU capitalizes on the temporal correlation in requests, and thus replaces the oldest object in the cache. LFU sorts objects based on their access frequency and evicts the object with the least frequency. GDS sorts objects based on a cost function and recency of requests, and evicts the one with the least value. We used object size as the cost function to allow GDS to maximize byte hit rate as indicated by [6]. LSB is designed for P2P traffic caching and it uses the number of transmitted bytes of an object as a sorting key. Objects which have transmitted the least amount of bytes will be evicted next. OPT looks at the *entire* stream of requests off-line and caches the objects that will serve the most number of bytes from the cache.

We measure the byte hit rate under several scenarios. First, we consider the case when objects requested by peers are downloaded entirely, that is, there are no aborted transactions. Then, we consider the case when the downloading peers prematurely terminate the downloads during the session, which is not uncommon in P2P systems. We also run our experiments under the independent reference model, where entries of the trace file are randomly shuffled to eliminate the effect of temporal correlation. We do that to isolate and study the effect of object popularity on caching. Finally, we run our experiments on the original trace with preserved temporal correlation between requests to study the combined effect of temporal correlation and popularity.

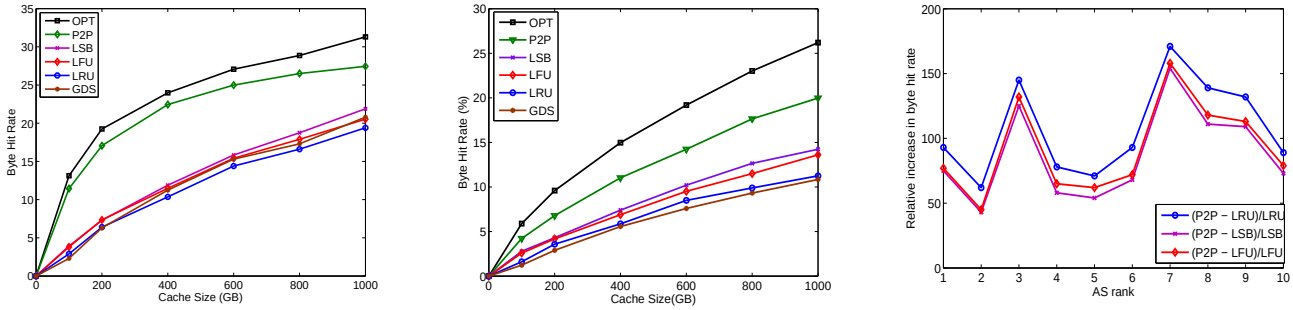
In all experiments, we use the ASes which have the most amount of traffic seen by our measurement node. This ensures that we have enough traffic from an AS to evaluate the effectiveness of deploying a cache for it.

### B. Caching under Full Downloads

One of the main advantages of our caching policy is that it effectively minimizes the impact of unpopular traffic by admitting objects incrementally. Traditional policies usually cache an entire object upon a miss, evicting other, perhaps more popular, objects if necessary. This may waste cache space by storing unpopular objects in the cache. To investigate the impact of partial versus full caching, we, first, assume that we have a no-failure scenario where peers download the object entirely. We look at all objects found in an AS, line them up in a randomly shuffled stream of requests and run them through the cache. We vary the cache size between 0 and 1000 GB.

Fig. 5 shows the byte hit rate for two representative ASes with different characteristics. These two ASes have different maximum achievable byte hit rates, which is defined as the fraction of traffic downloaded more than once, i.e., cacheable traffic, over the total amount of traffic. As shown in the figure, our policy outperforms other policies by as much as 200%. For instance, in AS397 (Fig. 5(a)) with a cache of 600 GB, our policy achieves a byte hit rate of 24%, which is almost double the rate achieved by LRU, LFU, GDS, and LSB policies. Moreover, the byte hit rate achieved by our algorithm is about 3% less than that of the optimal algorithm. Our trace shows that the amount of traffic seen in AS397 is around 24.9 tera bytes. This means that a reasonable cache of size 600 GB would have served about 6 tera bytes locally using our algorithm.

The reason traditional policies perform poorly for P2P traffic is because of the effect of unpopular objects. For example, one-timer objects are stored entirely under traditional policies on a miss. Under our policy, however, only one segment of each one-timer will find its way into the cache, thus minimizing their effect. The same could be said about 2nd-timers, 3rd timers and so on. Thus, our algorithm strives to discover the best objects to store in the cache by incrementally admitting them. Similar results were obtained for the other top ten ASes [21]. Our policy consistently preforms better than traditional policies. Fig. 5(c) summarizes the relative improvement in byte hit rate that our policy achieves over LRU, LFU, and LSB for the top ten ASes. The relative improvement is computed as the difference between the byte hit rate achieved by our policy and the byte hit rate achieved by another policy normalized by the byte hit rate of the other policy. For instance the improvement over LRU would be $(P2P - LRU)/LRU$. The figure shows a relative improvement of at least 40% and up to 180% can be achieved by using our algorithm. That is a significant gain given the large volume of the P2P traffic. We notice that the relative gain our policy achieves is larger in ASes with a substantial fraction of one-timers. We also observe that the achievable byte hit rate is around 15% to 40% with reasonable cache sizes. This is similar to the achievable byte hit rate for




(a) AS397, 48% of traffic is cacheable (b) AS95, 54% of traffic is cacheable (c) Top ten ASes

Fig. 5. Byte hit rate for different caching algorithms. No aborted downloads.

web caching, which is practically in the range 20%—35% (CISCO technical paper [25]), or as other sources indicate 30%—60% [26]. But due to the large size of P2P objects, a small byte hit rate amounts to savings of tera bytes of data.

As a final comment on Fig. 5, consider the absolute byte hit rate archived under our algorithm as well as the optimal byte hit rate in the two different ASes. Notice that although the percentage of cacheable traffic in AS397 is less than that of AS95, the absolute byte hit rate is higher in AS397. This is because popular objects in AS95 do not get as many requests as their counterparts in AS397. That is, the popularity distribution of AS95 has a more flattened head than that of AS397. We computed the skewness factor $\alpha$ and the plateau factor $q$ of the Mandelbrot-Zipf distribution that best fits the popularity distributions of these two ASes. We found that AS95 has $\alpha = 0.6$ and $q = 50$, while AS397 has $\alpha = 0.55$ and $q = 25$. Smaller $q$ values mean less flattened heads, and yield higher byte hit rates. Section V-C elaborates more on the impact of $\alpha$ and $q$ on the byte hit rate.

Next, we study the effect of temporal correlations on P2P caching by using the original unshuffled traces with preserved temporal correlations. Fig. 8 shows the byte hit rate under AS397. LRU and GDS perform slightly better than LSB and LFU because they make use of the temporal correlation between requests. However, the achieved byte hit rate under the four algorithms is still low compared to the byte hit rate achieved by our algorithm. Note that the absolute byte hit rate under our algorithm is slightly smaller than in the previous experiment where we used the independent reference model. This is because our algorithm does not consider temporal correlation. However, this reduction is small, less than 3%. The fact that the performance of our algorithm does not suffer much under temporal correlation and still outperforms other algorithms (e.g., LRU and GDS) could be explained as follows. We believe that object size is the dominant factor in caching for P2P systems, because the maximum cache size we used (1000 GB) is still small compared to the total size of objects, less than 5%—10% in most cases. As a consequence, object admission strategy is a key element in determining the byte hit rate, which our algorithm capitalizes on. Similar results for other ASes were obtained [21].

Due to the nature of P2P systems, peers could fail during a download, or abort a download. We run experiments to study the effect of aborted downloads on caching, and how robust our caching policy is. Following observations from [1], we allow 66% of downloads to abort anywhere in the download session. While our policy is designed to deal with aborted downloads, web replacement policies usually download the entire object upon a miss, and at times perform pre-fetching of popular objects. This is reasonable in the web since web objects are usually small, which means they take less cache space. But in P2P systems, objects are larger, and partial downloads constitute a significant number of the total number of downloads. Fig. 6 compares the byte hit rate with aborted downloads using several algorithms. Compared to the scenario of caching under full downloads (Section V-B), the performance of our algorithm improves slightly while the performance of other algorithms declines. The improvement in our policy could be explained by the fact that fewer bytes are missed in case of a failure. The performance of LRU, LFU, and LSB declines because they store an object upon a miss regardless of how much of it the client actually downloads. Hence, under aborted download scenarios, the byte hit rates for traditional policies suffers even more than it does under full download scenario. Similar results were obtained for the other top ten ASes. Our policy consistently outperforms LSB, LRU and LFU with a significant improvement margin in all top ten ASes. Fig. 7 shows that the relative improvement in byte hit rate is at least 50% and up to 200%.

### C. Effect of $\alpha$ and $q$ on P2P Caching

As we mention in Section III, P2P traffic can be modeled by a Mandelbrot-Zipf distribution with a skewness factor $\alpha$ between 0.4 and 0.70, and a plateau factor $q$ between between 5 and 60 In this section, we study the effect of $\alpha$ and $q$ on the byte hit rate of our algorithm via simulation. We did not use our traces because they may not capture the performance of our algorithm for all possible values of $\alpha$ and $q$. We randomly pick $100,000$ objects from our traces and generate their frequencies using Mandelbrot-Zipf with various values for $\alpha$ and $q$. We fix the cache size at 1,000 GB and we assume a no-failure model where peers download objects entirely.

To study the effect of different $\alpha$ values, we fix $q$ and change $\alpha$ between 0.4 and 1. As shown in Fig. 9, the byte hit rate

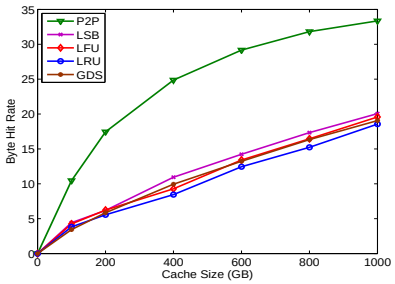



Fig. 6. Byte hit rate for different caching algorithms using traces with aborted downloads.

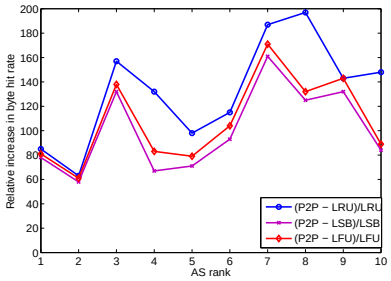



Fig. 7. Relative byte hit rate improvement.

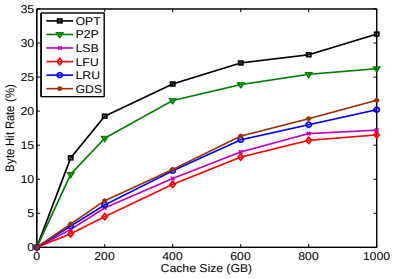



Fig. 8. Byte hit rate for different caching algorithms using traces with temporal correlations.

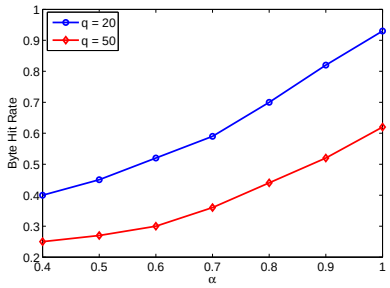



Fig. 9. The impact of $\alpha$ on caching.

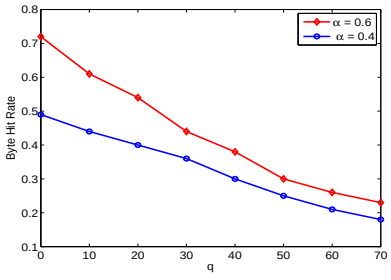



Fig. 10. The impact of $q$ on caching.

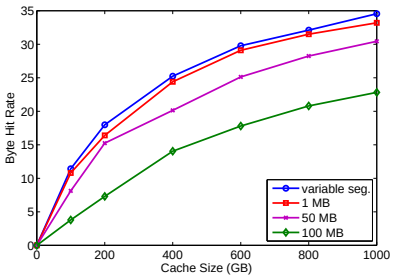



Fig. 11. The effect of segmentation on caching.

increases as the skewness factor $\alpha$ increases. This is intuitive since higher values of $\alpha$ mean that objects at the lowest ranks are more popular and caching them yields higher byte hit rate. Thus ASes whose popularity distribution is characterized with high $\alpha$ values would benefit more from caching than those with low $\alpha$ values.

Another parameter that determines the achievable byte hit rate is the plateau factor $q$ of the popularity distribution. $q$ reflects the flattened head we observed in Section III. Fig. 10 shows that the byte hit rate decreases as $q$ increases. The reason for this is that a higher value of $q$ means popular objects are requested less often. This has a negative impact on a cache that stores those popular objects, because they receive less downloads resulting in a decreased byte hit rate. Notice, though, that we are only exploring how changing the Mandelbrot-Zipf parameters impacts the caching performance and not suggesting using specific values for $q$ and $\alpha$. This means that there exists some ASes for which Mandelbrot-Zipf accurately captures the popularity model by a small value of $\alpha$ and a large value of $q$ resulting in popularity being spread out among objects more evenly, i.e., approaching a uniform distribution. But in such model all caching algorithms will suffer since the portion of requests received by popular objects decreases drastically.

### *D. Effect of Segmentation on P2P Caching*

Our algorithm uses a variable segmentation scheme whereby objects belonging to the same workload have the same segment size. As we discuss in Section IV, variable segmentation has several advantages over fixed segmentation. Using different segment size for different workloads helps reduce the overhead of managing too many segments if large objects were to have the same segment size as small objects. It also helps increase byte hit rate because larger segments for larger objects means that they will be available in the cache after relatively few requests. For example, if we use 1 MB segmentation, then an object of 4 GB will have 4000 segments to manage. On the other hand, using 10 MB segment size for would reduce the number of segments to 400 and still give a higher byte hit rate.

To measure the impact of segmentation on byte hit rate, we repeat the experiment of aborted downloads under AS397. Using different segment sizes, we, first, evaluate the byte hit rate when all objects have the same segment size (1 MB, 50 MB, and 100 MB) regardless of their size. Then we evaluate our variable segmentation scheme which we described in Section IV. As can be seen from Fig. 11, as the segment size increases, the byte hit rate degrades. This is because the cache evicts and admits segments only, and when a download aborts in the middle of downloading a segment, the rest of it will be unnecessarily stored. Similarly, under large segment sizes, we may admit more unpopular objects since admitting one segment of a one-timer amounts to admitting a substantial part of the object. The figure also shows that our variable segmentation scheme achieves similar byte hit rate as a uniform 1 MB segmentation, which imposes higher cache management overhead as discussed above.

## VI. Conclusion and Future Work

In this paper, we conducted a three-month measurement study on the Gnutella P2P system. Using real-world traces, we studied and modeled characteristics of P2P traffic that are

relevant to caching. We found that the popularity distribution of P2P objects cannot be captured accurately using Zipf-like distributions. We proposed a new Mandelbrot-Zipf model for P2P popularity and showed that it closely models object popularity in several AS domains. Our measurement study indicates that: (i) The Mandelbrot-Zipf popularity has a negative effect on hit rates of caches that use LRU or LFU policies, and (ii) Object admission strategies in P2P caching are as critical to cache performance as object replacement strategies.

We designed and evaluated a new P2P caching algorithm that is based on segmentation and incremental admission of objects according to their popularity. Using trace-driven simulations, we showed that our algorithm outperforms traditional algorithms by a significant margin and achieves a byte hit rate that is up to triple the byte hit rate achieved by other algorithms. Finally, because of aborted downloads, a caching policy should not perform pre-fetching of objects if its objective is to maximize byte hit rate, which is the strategy taken by our algorithm.

We are currently implementing our algorithm in the Squid [27] proxy cache and testing it using our traces. In the future, we intend to investigate preferential segment eviction, whereby segments of the same object carry different values.

## ACKNOWLEDGMENTS

We thank James Griffioen, the shepherd of our paper, and the anonymous reviewers for their constructive comments and suggestions. This research is partially supported by an NSERC Discovery Grant and by a President's Research Grant from Simon Fraser University .

## REFERENCES

[1] K. Gummadi, R. Dunn, S. Saroiu, S. Gribble, H. Levy, and J. Zahorjan, "Measurement, modeling, and analysis of a peer-to-peer file-sharing workload," in *Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, Bolton Landing, NY, USA, Oct. 2003, pp. 314–329.

[2] T. Karagiannis, A. Broido, N. Brownlee, K. C. Claffy, and M. Faloutsos, "Is P2P dying or just hiding?" in *Proc. of IEEE Global Telecommunications Conference (GLOBECOM'04)*, Dallas, TX, USA, Nov. 2004, pp. 1532–1538.

[3] T. Karagiannis, P. Rodriguez, and K. Papagiannaki, "Should internet service providers fear peer-assisted content distribution?" in *Proc. of the 5th ACM SIGCOMM Conference on Internet Measurement (IMC'05)*, Berkeley, CA, USA, Oct. 2005, pp. 63–76.

[4] S. Podlipnig and L. Bszrmenyi, "A survey of web cache replacement strategies," *ACM Computing Surveys*, vol. 35, no. 4, pp. 347–398, Dec. 2003.

[5] J. Liu and J. Xu, "Proxy caching for media streaming over the Internet," *IEEE Communications Magazine*, vol. 42, no. 8, pp. 88–94, Aug. 2004.

[6] P. Cao and S. Irani, "Cost-aware WWW proxy caching algorithms," in *Proc. of USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, USA, Dec 1997, pp. 193–206.

[7] A. Wierzbicki, N. Leibowitz, M. Ripeanu, and R. Wozniak, "Cache replacement policies revisited: The case of P2P traffic," in *Proc. of the 4th International Workshop on Global and Peer-to-Peer Computing (GP2P'04)*, Chicago, IL, USA, Apr. 2004, pp. 182–189.

[8] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and Zipf-like distributions: Evidence and implications," in *Proc. of INFOCOM'99*, New York, NY, USA, Mar. 1999, pp. 126–134.

[9] S. Sen and J. Wang, "Analyzing peer-to-peer traffic across large networks," *IEEE/ACM Transactions on Networking*, vol. 12, no. 2, pp. 219–232, Apr. 2004.

[10] A. Klemm, C. Lindemann, M. K. Vernon, and O. P. Waldhorst, "Characterizing the query behavior in peer-to-peer file sharing systems," in *Proc. of ACM/SIGCOMM Internet Measurement Conference (IMC'04)*, Taormina, Sicily, Italy, Oct. 2004, pp. 55–67.

[11] N. Leibowitz, A. Bergman, R. Ben-Shaul, and A. Shavit, "Are file swapping networks cacheable? Characterizing P2P traffic," in *Proc. of the 7th International Workshop on Web Content Caching and Distribution (WCW'02)*, Boulder, CO, USA, Aug. 2002.

[12] S. Jin and A. Bestavros, "Popularity-Aware GreedyDual-Size Web Proxy Caching Algorithms," in *Proc. of the IEEE International Conference on Distributed Computing Systems (ICDCS'00)*, Taiwan, May 2000.

[13] S. Jin, A. Bestavros, and A. Iyengar, "Network-Aware Partial Caching For Internet Streaming Media," *ACM Multimedia Systems Journal*, vol. 9, no. 4, pp. 386–396, Oct. 2003.

[14] S. Park, E. Lim, and K. Chung, "Popularity-based Partial Caching for VOD Systems using a Proxy Server." in *Proc. of the 15th International Parallel & Distributed Processing Symposium (IPDPS'01)*, San Francisco, CA, USA, Apr. 2001.

[15] "Gnutella Home Page," http://www.gnutella.com.

[16] "Limewire Home Page," http://www.limewire.com/.

[17] S. Saroiu, P. K. Gummadi, and S. D. Gribble, "Measuring and analyzing the characteristics of Napster and Gnutella hosts," *ACM/Springer Multimedia Systems Journal*, vol. 9, no. 2, pp. 170–184, Aug. 2003.

[18] J. Pouwelse, P. Garbacki, D. Epema, and H. Sips, "The BitTorrent P2P file-sharing system: Measurements and analysis," in *Proc. of the 4th International Workshop on Peer-To-Peer Systems (IPTPS'05)*, Ithaca, NY, USA, Feb. 2005, pp. 205–216.

[19] "Network Systems Lab Home Page," http://nsl.cs.surrey.sfu.ca.

[20] "GeoIP Database Home Page," http://www.maxmind.com.

[21] O. Saleh and M. Hefeeda, "Modeling and caching of peer-to-peer traffic," Simon Fraser University, Tech. Rep. TR 2006-11, May 2006.

[22] Z. Silagadze, "Citations and the Zipf-Mandelbrot's law," *Complex Systems*, vol. 11, no. 487–499, 1997.

[23] "Emule Project Home Page," http://www.emule-project.net/.

[24] "BearShare Home Page," http://www.bearshare.com/.

[25] G. Huston, "Web Caching," *The Internet Protocol Journal*, vol. 2, no. 3, Sept. 1999.

[26] M. Rabinovich and O. Spatscheck, *Web Caching and Replication*. Addison-Wesley, 2002.

[27] "Squid Web Proxy Cache Home Page," www.squid-cache.org/.