Design and Evaluation of a Proxy Cache for Peer-to-Peer Traffic

Mohamed Hefeeda, *Senior Member*, *IEEE*, Cheng-Hsin Hsu, *Member*, *IEEE*, and Kianoosh Mokhtarian, *Student Member*, *IEEE*

Abstract—Peer-to-peer (P2P) systems generate a major fraction of the current Internet traffic, and they significantly increase the load on ISP networks and the cost of running and connecting customer networks (e.g., universities and companies) to the Internet. To mitigate these negative impacts, many previous works in the literature have proposed caching of P2P traffic, but very few (if any) have considered designing a caching system to actually do it. This paper demonstrates that caching P2P traffic is more complex than caching other Internet traffic, and it needs several new algorithms and storage systems. Then, the paper presents the design and evaluation of a complete, running, proxy cache for P2P traffic, called pCache. pCache transparently intercepts and serves traffic from different P2P systems. A new storage system is proposed and implemented in pCache. This storage system is optimized for storing P2P traffic, and it is shown to outperform other storage systems. In addition, a new algorithm to infer the information required to store and serve P2P traffic by the cache is proposed. Furthermore, extensive experiments to evaluate all aspects of pCache using actual implementation and real P2P traffic are presented.

Index Terms—Caching, peer-to-peer systems, file sharing, storage systems, performance evaluation.

1 Introduction

File sharing using peer-to-peer (P2P) systems is currently the killer application for the Internet. A huge amount of traffic is generated daily by P2P systems [1], [2], [3]. This huge amount of traffic costs university campuses thousands of dollars every year. Internet service providers (ISPs) also suffer from P2P traffic [4], because it increases the load on their routers and links. Some ISPs shape or even block P2P traffic to reduce the cost. This may not be possible for some ISPs, because they fear losing customers to their competitors. To mitigate the negative effects of P2P traffic, several approaches have been proposed in the literature, such as designing locality-aware neighbor selection algorithms [5] and caching of P2P traffic [6], [7], [8]. We believe that caching is a promising approach to mitigate some of the negative consequences of P2P traffic, because objects in P2P systems are mostly immutable [1] and the traffic is highly repetitive [9]. In addition, caching does not require changing P2P protocols and can be deployed transparently from clients. Therefore, ISPs can readily deploy caching systems to reduce their costs. Furthermore, caching can coexist with other approaches, e.g., enhancing P2P traffic locality, to address the problems created by the enormous volume of P2P traffic.

- *M. Hefeeda is with the School of Computing Science, Simon Fraser University Surrey, 250-13450 102nd Ave, Surrey, BC V3T 0A3, Canada. E-mail: mhefeeda@cs.sfu.ca.*
- *C.-H. Hsu is with Deutsche Telekom R&D Lab USA, 5050 El Camino Real, Suite 221, Los Altos, CA 94022. E-mail: cheng-hsin.hsu@telekom.com.*
- *K. Mokhtarian is with the Electrical and Computer Engineering Department, University of Toronto, 10 King's College Road, Toronto, ON M5S 3G4, Canada. Email: kianoosh.mokhtarian@utoronto.ca.*

Manuscript received 8 June 2009; revised 2 Dec. 2009; accepted 8 Jan. 2010; published online 23 Feb. 2011.
Recommended for acceptance by Y. Yang.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2009-06-0265.
Digital Object Identifier no. 10.1109/TC.2011.57.

In this paper, we present the design, implementation, and evaluation of a proxy cache for P2P traffic, which we call pCache. pCache is designed to transparently intercept and serve traffic from different P2P systems, while not affecting traffic from other Internet applications. pCache explicitly considers the characteristics and requirements of P2P traffic. As shown by numerous previous studies, e.g., [1], [2], [8], [10], [11], network traffic generated by P2P applications has different characteristics than traffic generated by other Internet applications. For example, object size, object popularity, and connection pattern in P2P systems are quite different from their counterparts in web systems. Most of these characteristics impact the performance of caching systems, and therefore, they should be considered in their design. In addition, as will be demonstrated in this paper, designing proxy caches for P2P traffic is more complex than designing caches for web or multimedia traffic, because of the multitude and diverse nature of existing P2P systems. Furthermore, P2P protocols are not standardized and their designers did not provision for the potential caching of P2P traffic. Therefore, even deciding whether a connection carries P2P traffic and if so extracting the relevant information for the cache to function (e.g., the requested byte range) are nontrivial tasks.

The main contributions of this paper are as follows:

- We propose a new storage system optimized for P2P proxy caches. The proposed system efficiently serves requests for object segments of arbitrary lengths, and it has a dynamic segment merging scheme to minimize the number of disk I/O operations. Using traces collected from a popular P2P system, we show that the proposed storage system is much more efficient in handling P2P traffic than storage systems designed for web proxy caches. More specifically,

our experimental results indicate that the proposed storage system is about five times more efficient than the storage system used in a famous web proxy implementation.

- We propose a new algorithm to infer the information required to store and serve P2P traffic by the cache. This algorithm is needed because some P2P systems (e.g., BitTorrent) maintain information about the exchanged objects in metafiles that are held by peers, and peers request data relative to information in these metafiles, which are not known to the cache. Our inference algorithm is efficient and provides a quantifiable confidence on the returned results. Our experiments with real BitTorrent clients running behind our pCache confirm our theoretical analysis and show that the proposed algorithm returns correct estimates in more than 99.7 percent of the cases. Also, the ideas of the inference algorithm are general and could be useful in inferring other characteristics of P2P traffic.
- We demonstrate the importance and difficulty of providing full transparency in P2P proxy caches, and we present our solution for implementing it in the Linux kernel. We also propose efficient splicing of non-P2P connections in order to reduce the processing and memory overhead on the cache.
- We conduct an extensive experimental study to evaluate the proposed pCache system and its individual components using actual implementation in two real P2P networks: BitTorrent and Gnutella which are currently among the most popular P2P systems. (A recent white paper indicates the five most popular P2P protocols are: BitTorrent, eDonkey, Gnutella, DirectConnect, and Ares [12]). BitTorrent and Gnutella are chosen because they are different in their design and operation. The former is swarm-based and uses built-in incentive schemes, while the latter uses a two-tier overlay network. Our results show that our pCache is scalable and it benefits both the clients and the ISP in which it is deployed, without hurting the performance of the P2P networks. Specifically, clients behind the cache achieve much higher download speeds than other clients running in the same conditions without the cache. In addition, a significant portion of the traffic is served from the cache, which reduces the load on the expensive WAN links for the ISP. Our results also show that the cache does not reduce the connectivity of clients behind it, nor does it reduce their upload speeds. This is important for the whole P2P network, because reduced connectivity could lead to decreased availability of peers and the content stored on them, while reduced upload speeds could degrade the P2P network scalability.

In addition, this paper summarizes our experiences and lessons learned during designing and implementing a running system for caching P2P traffic, which was demonstrated in [13]. These lessons could be of interest to other researchers and to companies developing products for managing P2P traffic. We make the source code of pCache (more than 11,000 lines of C++ code) and our P2P traffic traces available to the research community [14]. Because it is open source, pCache could stimulate more research on developing methods for effective handling of P2P traffic in order to reduce its negative impacts on ISPs and the Internet.

The rest of this paper is organized as follows: In Section 2, we summarize the related work. Section 3 presents an overview of the proposed pCache system. This is followed by separate sections describing different components of pCache. Section 4 presents the new inference algorithm. Section 5 presents the proposed storage management system. Section 6 explains why full transparency is required in P2P proxy caches and describes our method to achieve it in pCache. It also explains how non-P2P connections are efficiently tunneled through pCache. We experimentally evaluate the performance of pCache in Section 7. Finally, in Section 8, we conclude the paper and outline possible extensions for pCache.

2 Related Work

2.1 P2P Traffic Caching: Models and Systems

The benefits of caching P2P traffic have been shown in [9] and [4]. Object replacement algorithms especially designed for proxy cache of P2P traffic have also been proposed in [6] and in our previous works [8], [11]. The above works do not present the design and implementation of an actual proxy cache for P2P traffic, nor do they address storage management issues in such caches. The authors of [7] propose using already-deployed web caches to serve P2P traffic. This, however, requires modifying the P2P protocols to wrap their messages in HTTP format and to discover the location of the nearest web cache. Given the distributed and autonomous nature of the communities developing P2P client software, incorporating these modifications into actual clients may not be practical. In addition, several measurement studies have shown that the characteristics of P2P traffic are quite different from those of web traffic [1], [2], [8], [10], [11]. These different characteristics indicate that web proxy caches may yield poor performance if they were to be used for P2P traffic. The experimental results presented in this paper confirm this intuition.

In order to store and serve P2P traffic, proxy caches need to identify connections belonging to P2P systems and extract needed information from them. Several previous works address the problem of identifying P2P traffic, including [15], [16], [17]. The authors of [15] identify P2P traffic based on application signatures appearing in the TCP stream. The authors of [17] analyze the behavior of different P2P protocols, and identify patterns specific to each protocol. The authors of [16] identify P2P traffic using only transport layer information. A comparison among different methods is conducted in [18]. Unlike our work, all previous identification approaches only detect the presence of P2P traffic: they just decide whether a packet or a stream of packets belongs to one of the known P2P systems. This is useful in applications such as traffic shaping and blocking, capacity planning, and service differentiation. P2P traffic caching, however, does need to go beyond just identifying P2P traffic; it requires not only the exchanged object ID, but also the requested byte range of that object. In some P2P protocols, this information can easily be obtained by

reading a few bytes from the payload, while in others it is not straightforward.

Several P2P caching products have been introduced to the market. Oversi's OverCache [19] implements P2P caching, but takes a quite different approach compared to pCache. An OverCache server participates in P2P networks and only serves peers within the ISP. This approach may negatively affect fairness in P2P networks because peers in ISPs with OverCache deployed would never contribute as they can always receive data from the cache servers without uploading to others. This in turn degrades the performance of P2P networks. PeerApp's UltraBand [20] supports multiple P2P protocols, such as BitTorrent, Gnutella, and FastTrack. These commercial products demonstrate the importance and timeliness of the problem addressed in this paper. The details of these products, however, are not publicly available, thus the research community cannot use them to evaluate new P2P caching algorithms. In contrast, the pCache system is open source and could be used to develop algorithms for effective handling of P2P traffic in order to reduce loads on ISPs and the Internet. We develop pCache software as a proof of concept, rather than yet another commercial P2P cache. Therefore, we do not compare our pCache against these commercial products.

2.2 Storage Management for Proxy Caches

Several storage management systems have been proposed in the literature for web proxy caches [21], [22], [23], [24] and for multimedia proxy caches [25], [26], [27]. These systems offer services that may not be useful for P2P proxy caches, e.g., minimizing start-up delay and clustering of multiple objects coming from the same web server, and they lack other services, e.g., serving partial requests with arbitrary byte ranges, which are essential in P2P proxy caches. We describe some examples to show why such systems are not suitable for P2P caches.

Most storage systems of web proxy caches are designed for small objects. For example, the Damelo system [28] supports objects that have sizes less than a memory page, which is rather small for P2P systems. The UCFS system [23] maintains data structures to combine several tiny web objects together in order to fill a single disk block and to increase disk utilization. This adds overhead and is not needed in P2P systems, because a segment of an object is typically much larger than a disk block. Clustering of web objects downloaded from the same web server is also common in web proxy caches [22], [24]. This clustering exploits the temporal correlation among these objects in order to store them near to each other on the disk. This clustering is not useful in P2P systems, because even a single object is typically downloaded from multiple senders.

Proxy caches for multimedia streaming systems [25], [26], [27], on the other hand, could store large objects and serve byte ranges. Multimedia proxy caches can be roughly categorized into four classes [25]: sliding-interval, prefix, segment-based, and rate-split caching. Sliding-interval caching employs a sliding-window for each cached object to take advantage of the sequential access pattern that is common in multimedia systems. Sliding-interval caching is not applicable to P2P traffic, because P2P applications do not request segments in a sequential manner. Prefix caching stores the initial portion of multimedia objects to minimize client start-up delays. P2P applications seek shorter total download times rather than start-up delays. Segment-based multimedia caching divides a multimedia object into segments using segmentation strategies, such as uniform, exponential, and frame-based. P2P proxy caches do not have the freedom to choose a segmentation strategy; it is imposed by P2P software clients. Rate-split caching employs scalable video coding that encodes a video into several substreams, and selectively caches some of these substreams. This requires scalable video coding structures, which renders rate-split caching useless for P2P applications.

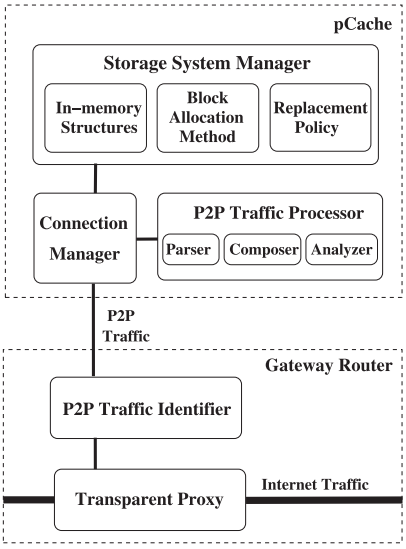

Fig. 1. The design of pCache.

3 Overview of pCache

The proposed pCache is to be used by autonomous systems (ASes) or ISPs that are interested in reducing the burden of P2P traffic. Caches in different ASes work independently from each other; we do not consider cooperative caching in this paper. pCache would be deployed at or near the gateway router of an AS. The main components of pCache are illustrated in Fig. 1. At high level, a client participating in a P2P network issues a request to download an object. This request is transparently intercepted by pCache. If the requested object or parts of it are stored in the cache, they are served to the requesting client. This saves bandwidth on the external (expensive) links to the Internet. If no part of the requested object is found in the cache, the request is forwarded to the P2P network. When the response comes back, pCache may store a copy of the object for future requests from other clients in its AS. Clients inside the AS as well as external clients are not aware of pCache, i.e., pCache is fully transparent in both directions.

As shown in Fig. 1, the Transparent Proxy and P2P Traffic Identifier components reside on the gateway router. They transparently inspect traffic going through the router and forward only P2P connections to pCache. Traffic that does not belong to any P2P system is processed by the router in the regular way and is not affected by the presence of pCache. Once a connection is identified as belonging to a P2P system, it is passed to the Connection Manager, which

coordinates different components of pCache to store and serve requests from this connection. pCache has a custom-designed Storage System optimized for P2P traffic. In addition, pCache needs to communicate with peers from different P2P systems. For each supported P2P system, the P2P Traffic Processor provides three modules to enable this communication: Parser, Composer, and Analyzer. The Parser performs functions such as identifying control and payload messages, and extracting messages that could be of interest to the cache such as object request messages. The Composer constructs properly formatted messages to be sent to peers. The Analyzer is a placeholder for any auxiliary functions that may need to be performed on P2P traffic from some systems. For example, in BitTorrent, the Analyzer infers information (piece length) needed by pCache that is not included in messages exchanged between peers.

The design of the proposed P2P proxy cache is the result of several iterations and refinements based on extensive experimentation. Given the diverse and complex nature of P2P systems, proposing a simple, well-structured, design that is extensible to support various P2P systems is indeed a nontrivial systems research problem. Our running prototype currently serves BitTorrent and Gnutella traffic at the same time. To support a new P2P system, two things need to be done: 1) installing the corresponding application identifier in the P2P Traffic Identifier, and 2) loading the appropriate Parser, Composer, and optionally Analyzer modules of the P2P Traffic Processor. Both can be done in runtime without recompiling or impacting other parts of the system.

Finally, we should mention that the proxy cache design in Fig. 1 does *not* require users of P2P systems to perform any special configurations of their client software, nor does it need the developers of P2P systems to cooperate and modify any parts of their protocols. It caches and serves current P2P traffic as is. Our design, however, can further be simplified to support cases in which P2P users and/or developers may *cooperate* with ISPs for mutual benefits—a trend that has recently seen some interests with projects, such as P4P [29], [30]. For example, if users were to configure their P2P clients to use proxy caches in their ISPs listening on a specific port, the P2P Traffic Identifier would be much simpler.

4 P2P Traffic Identifier and Processor

This section describes the P2P Traffic Identifier and Processor components of pCache, and presents a new algorithm to infer information needed by the cache to store and serve P2P traffic.

4.1 Overview

The P2P Traffic Identifier determines whether a connection belongs to any P2P system known to the cache. This is done by comparing a number of bytes from the connection stream against known P2P application signatures. The details are similar to the work in [15], and therefore omitted. We have implemented identifiers for BitTorrent and Gnutella.

To store and serve P2P traffic, the cache needs to perform several functions beyond identifying the traffic. These functions are provided by the P2P Traffic Processor, which has three components: Parser, Composer, and Analyzer. By inspecting the byte stream of the connection, the Parser determines the boundaries of messages exchanged between peers, and it extracts the request and response messages that are of interest to the cache. The Parser returns the ID of the object being downloaded in the session, as well as the requested byte range (start and end bytes). The byte range is relative to the whole object. The Composer prepares protocol-specific messages, and may combine data stored in the cache with data obtained from the network into one message to be sent to a peer. While the Parser and Composer are mostly implementation details, the Analyzer is not. The Analyzer contains an algorithm to *infer* information required by the cache to store and serve P2P traffic, but is not included in the traffic itself. This is not unusual (as demonstrated below for the widely deployed BitTorrent), since P2P protocols are not standardized and their designers did not conceive/provision for caching P2P traffic. We propose an algorithm to infer this information with a *quantifiable confidence*.

4.2 Inference Algorithm for Information Required by the Cache

The P2P proxy cache requires specifying the requested byte range such that it can correctly store and serve segments. Some P2P protocols, most notably BitTorrent, do not provide this information in the traffic itself. Rather, they indirectly specify the byte range, relative to information held only by end peers and not known to the cache. Thus, the cache needs to employ an algorithm to infer the required information, which we propose in this section. We describe our algorithm in the context of the BitTorrent protocol, which is currently the most widely used P2P protocol [12]. Nonetheless, the ideas and analysis of our algorithm are fairly general and could be applied to other P2P protocols, after collecting the appropriate statistics and making minor adjustments.

Objects exchanged in BitTorrent are divided into equal-size *pieces*. A single piece is downloaded by issuing multiple requests for byte ranges, i.e., segments, within the piece. Thus, a piece is composed of multiple segments. In request messages, the requested byte range is specified by an offset within the piece and the number of bytes in the range. Peers know the length of the piece of the object being downloaded, because it is included in the metafile (torrent file) held by them. The cache needs the piece length for three reasons. First, the piece length is needed to perform segment merging, which can reduce the overhead on the cache. For example, assuming the cache has received byte range $[0, 16]$ KB of piece 1 and range $[0, 8]$ KB of piece 2; without knowing the precise piece length, the cache cannot merge these two byte ranges into a continuous byte range. Second, the piece length is required to support cross-torrent caching of the same content, as different torrent files can have different piece length. Third, the piece length is needed for cross-system caching. For example, a file cached for in BitTorrent networks may be used to serve Gnutella users requesting the same file, while Gnutella protocol has no concept of piece length. One way for the cache to obtain this information is to capture metafiles and match them with objects. This, however, is not always possible, because peers frequently receive metafiles by various means/applications including e-mails and web downloads.

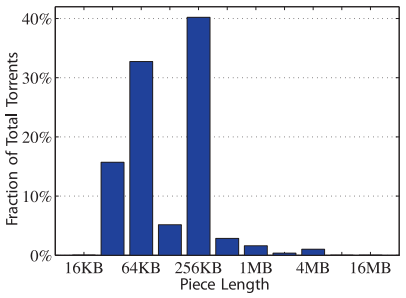

Fig. 2. Piece length distribution in BitTorrent.

Our algorithm infers the piece length of an object when the object is first seen by the cache. The algorithm monitors a few segment requests, and then it makes an informed guess about the piece length. To design the piece length inference algorithm, we collected statistics on the distribution of the piece length in actual BitTorrent objects. We wrote a script to gather and analyze a large number of torrent files. We collected more than 43,000 unique torrent files from popular torrent websites on the Internet. Our analysis revealed that more than 99.8 percent of the pieces have sizes that are power of 2. The histogram of the piece length distribution is given in Fig. 2, which we will use in the algorithm. Using these results, we define the set of all possible piece lengths as $\mathcal{S} = \{2^{k_{min}}, 2^{k_{min}+1}, \ldots, 2^{k_{max}}\}$, where $2^{k_{min}}$ and $2^{k_{max}}$ are the minimum and maximum possible piece lengths. We denote by L the actual piece length that we are trying to infer. A request for a segment comes in the form $\{offset, size\}$ within this piece, which tells us that the piece length is at least as large as $offset + size$. There are multiple requests within the piece. We denote each request by $x_i = offset_i + size_i$, where $0 < x_i \leq L$. The inference algorithm observes a set of n samples $x_1, x_2, \ldots, x_n$ and returns a reliable guess for L.

After observing the n-th sample, the algorithm computes k such that 2^k is the minimum power of two integer that is greater than or equal to x_i $(i = 1, 2, \ldots, n)$. For example, if $n = 5$ and the x_i values are 9, 6, 13, 4, 7, then the estimated piece length would be $2^k = 16$. Our goal is to determine the minimum number of samples n such that the probability that the estimated piece length is correct, i.e., $Pr(2^k = L)$, exceeds a given threshold, say 99 percent. We compute the probability that the estimated piece length is correct as follows: We define E as the event that the n samples are in the range $[0, 2^k]$ and at least one of them does not fall in $[0, 2^{k-1}]$, where 2^k is the estimated value returned by the algorithm. Assuming that each sample is equally likely to be anywhere within the range $[0, L]$, we have

$$\begin{aligned} Pr(2^k = L \mid E) &= Pr(E \mid 2^k = L)\frac{Pr(2^k = L)}{Pr(E)} \\ &= [1 - Pr(all\ n\ samples\ in\ [0, 2^{k-1}] \mid 2^k = L)]\frac{Pr(2^k = L)}{Pr(E)} \\ &= \left[1 - \left(\frac{1}{2}\right)^n\right]\frac{Pr(2^k = L)}{Pr(E)}. \end{aligned} \tag{1}$$

In the above equation, $Pr(2^k = L)$ can be directly known from the empirically derived histogram in Fig. 2. Furthermore, $Pr(E)$ is given by

$$\begin{aligned} Pr(E) &= \sum_{l \in \mathcal{S}} Pr(l = L)Pr(E|l = L) \\ &= Pr(2^k = L)\left(1 - \left(\frac{1}{2}\right)^n\right) + Pr(2^{k+1} = L)\left(\left(\frac{1}{2}\right)^n - \left(\frac{1}{4}\right)^n\right) \\ &+ \cdots + Pr(2^{k_{max}} = L)\left(\left(\left(\frac{1}{2}\right)^{k_{max}-k}\right)^n - \left(\left(\frac{1}{2}\right)^{k_{max}-k+1}\right)^n\right). \end{aligned} \tag{2}$$

Our algorithm solves (2) and (1) using the histogram in Fig. 2 to find the required number of samples n in order to achieve a given probability of correctness.

The assumption that samples are equally likely to be in $[0, L]$ is realistic, because the cache observes requests from many clients at the same time for an object, and the clients are not synchronized. In few cases, however, there could be one client and that client may issue requests sequentially. This depends on the actual implementation of the BitTorrent client software. To address this case, we use the following heuristic. We maintain the maximum value seen in the samples taken so far. If a new sample increases this maximum value, we reset the counters of the observed samples. In Section 7.5, we empirically validate the above inference algorithm and we show that the simple heuristic improves its accuracy.

Handling incorrect inferences. In the evaluation section, we experimentally show that the accuracy of our inference algorithm is about 99.7 percent when we use six segments in the inference, and it is almost 100 percent if we use 10 segments. The very rare incorrect inferences are observed and handled by the cache as follows. An incorrect inference is detected by observing a request exceeding the inferred piece boundary, because we use the minimum possible estimate for the piece length. This may happen at the beginning of a download session for an object that was never seen by the cache before. The cache would have likely stored very few segments of this object and most of them are still in the buffer, not flushed to the disk yet. Thus, all the cache needs to do is to readjust a few pointers in the memory with the correct piece length. In the worst case, a few disk blocks will also need to be moved to other disk locations. An even simpler solution is possible: discard from the cache these few segments, which has a negligible cost. Finally, we note that the cache starts storing segments after the inference algorithm returns an estimate, but it does not immediately serve these segments to other clients until a sufficient number of segments (e.g., 100) have been downloaded to make the probability of incorrect inference practically zero. Therefore, the consequence of a rare incorrect inference is a slight overhead in the storage system and for a temporary period (till the correct piece length is computed), and no wrong data are served to the clients at anytime.

5 Storage Management

In this section, we elaborate on the need for a new storage system, in addition to what we mentioned in Section 2. Then, we present the design of the proposed storage system.

5.1 The Need for a New Storage System

In P2P systems, a receiving peer requests an object from multiple sending peers in the form of segments, where a segment is a range of bytes. Object segmentation is protocol-dependent and even implementation-dependent. Moreover, segment sizes could vary based on the number and type of senders in the download session, as in the case of Gnutella. Therefore, successive requests of the same object can be composed of segments with different sizes. For example, a request comes to the cache for the byte range [128-256 KB] as one segment, which is then stored locally in the cache for future requests. However, a later request may come for the byte range [0-512 KB] of the same object and again as one segment. The cache should be able to identify and serve the cached portion of the second request. Furthermore, previous work [11] showed that to improve the cache performance, objects should be incrementally admitted in the cache because objects in P2P systems are fairly large, their popularity follows a flattened head model [1], [11], and they may not be even downloaded in their entirety since users may abort the download sessions [1]. This means that the cache will usually store random fragments of objects, not complete contiguous objects, and the cache should be able to serve these partial objects.

While web proxy caches share some characteristics with P2P proxy caches, their storage systems can yield poor performance for P2P proxy caches, as we will show in Section 7.3. The most important reason is that most web proxy caches consider objects with unique IDs, such as URLs, in their entirety. P2P applications almost never request entire objects, instead, segments of objects are exchanged among peers. A simple treatment of reusing web proxy caches for P2P traffic is to define a hash function on both object ID and segment range, and consider segments as independent entities. This simple approach, however, has two major flaws. First, the hash function destroys the correlation among segments belonging to the same objects. Second, this approach cannot support partial hits, because different segment ranges are hashed to different, unrelated, values.

The proposed storage system supports efficient lookups for partial hits. It also improves the scalability of the cache by reducing the number of I/O operations. This is done by preserving the correlation among segments of the same objects, and dynamically merging segments. To the best of our knowledge, there are no other storage systems proposed in the literature for P2P proxy caches, even though the majority of the Internet traffic comes from P2P systems [1], [2], [3].

5.2 The Proposed Storage System

A *simplified* view of the proposed storage system is shown in Fig. 3. We implement the proposed system in the user space, so that it is not tightly coupled with the kernels of operating systems, and can be easily ported to different kernel versions, operating system distributions, and even to various operating systems. As indicated by [24], user-space storage systems yield very close performance to the kernel-space ones. A user-space storage management system can be built on top of a large file created by the file system of the underlying operating system, or it can be built directly on a raw disk partition. We implement and analyze two versions

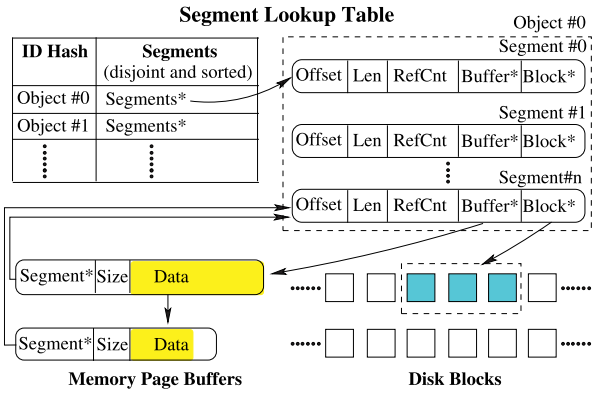

Fig. 3. The proposed storage management system.

of the proposed storage management system: on top of the ext2 Linux file system (denoted by pCache/Fs) and on a raw disk partition (denoted by pCache/Raw).

As shown in Fig. 3, the proposed storage system maintains two structures in the memory: metadata and page buffers. The metadata represent a two-level lookup table designed to enable efficient segment lookups. The first level is a hash table keyed on object IDs; collisions are resolved using common chaining techniques. Every entry points to the second level of the table, which is a set of cached segments belonging to the same object. Every segment entry consists of a few fields: Offset indicates the absolute segment location within the object, Len represents the number of bytes in this segment, RefCnt keeps track of how many connections are currently using this segment, Buffer points to the allocated page buffers, and Block points to the assigned disk blocks. Each disk is divided into fixed-size disk blocks, which are the smallest units of disk operations. Therefore, block size is a system parameter that may affect caching performance: larger block sizes are more vulnerable to segmentations, while smaller block sizes may lead to high space overhead. RefCnt is used to prevent evicting a buffer page if there are connections currently using it.

Notice that segments do not arrive to the cache sequentially, and not all segments of an object will be stored in the cache. Thus, a naive contiguous allocation of all segments will waste memory, and will not efficiently find partial hits. We implement the set of cached segments as a balanced (red-black) binary tree, which is sorted based on the Offset field. Using this structure, partial hits can be found in at most $O(\log S)$ steps, where S is the number of segments in the object. This is done by searching on the offset field. Segment insertions and deletions are also done in logarithmic number of steps. Since this structure supports partial hits, the cached data are never obtained from the P2P network again, and only mutually disjoint segments are stored in the cache.

The second part of the in-memory structures is the page buffers. Page buffers are used to reduce disk I/O operations as well as to perform segment merging. As shown in Fig. 3, we propose to use multiple sizes of page buffers, because requests come to the cache from different P2P systems in variable sizes. We also preallocate these pages in memory for efficiency. Dynamic memory allocation may not be suitable for proxy caches, since it imposes processing

overheads. We maintain unoccupied pages of the same size in the same free page list. If peers request segments that are in the buffers, they are served from memory and no disk I/O operations are issued. If the requested segments are on the disk, they need to be swapped in some free memory buffers. When all free buffers are used up, the least popular data in some of the buffers are swapped out to the disk if these data have been modified since they were brought in memory, and they are overwritten otherwise.

6 Transparent Connection Management

This section starts by clarifying the importance and complexity of providing full transparency in P2P proxy caches. Then, it presents our proposed method for splicing non-P2P connections in order to reduce the processing and memory overhead on the cache. We implement the Connection Manager in Linux 2.6. The details on the implementation of connection redirection and the operation of the Connection Manager [14] are not presented due to the space limitations.

6.1 Importance of Full Transparency

Based on our experience of developing a running caching system, we argue that full transparency is important in P2P proxy caches. This is because nontransparent proxies may not take full advantage of the deployed caches, since they require users to manually configure their applications. This is even worse in the P2P traffic caching case due to the existence of multiple P2P systems, where each has many different client implementations. Transparent proxies, on the other hand, actively intercept connections between internal and external hosts, and they do not require error-prone manual configurations of the software clients.

Transparency in many web caches, e.g., Squid [31], is achieved as follows. The cache intercepts the TCP connection between a local client and a remote web server. This interception is done by connection redirection, which forwards connection setup packets traversing through the gateway router to a proxy process. The proxy process accepts this connection and serves requests on it. This may require the proxy to create a connection with the remote server. The web cache uses its IP address when communicating with the web server. Thus, the server can detect the existence of the cache and needs to communicate with it. We call this type of caches partially transparent, because the remote server is aware of the cache. In contrast, we refer to proxies that do not reveal their existence as fully transparent proxies. When a fully transparent proxy communicates with the internal host, it uses the IP address and port number of the external host, and similarly when it communicates with the external host it uses the information of the internal host.

While partial transparency is sufficient for most web proxy caches, it is not enough and will not work for P2P proxy caches. This is because external peers may not respond to requests coming from the IP address of the cache, since the cache is not part of the P2P network and it does not participate in the P2P protocol. Implementing many P2P protocols in the cache to make it participate in P2P networks is a tedious task. More important, participation in P2P networks imposes significant overhead on the cache itself, because it will have to process protocol messages and potentially serve too many objects to external peers. For example, if the cache was to participate in the tit-for-tat BitTorrent network, it would have to upload data to other external peers proportional to data downloaded by the cache on behalf of *all* internal BitTorrent peers. Furthermore, some P2P systems require registration and authentication steps that must be done by users, and the cache cannot do these steps.

Supporting full transparency in P2P proxy caches is not trivial, because it requires the cache to process and modify packets destined to remote peers, i.e., its network stack accepts packets with nonlocal IP addresses and port numbers.

6.2 Efficient Splicing of Non-P2P Connections

Since P2P systems use dynamic ports, the proxy process may initially intercept some connections that do not belong to P2P systems. This can only be discovered after inspecting a few packets using the P2P Traffic Identification module. Each intercepted connection is split into a pair of connections, and all packets have to go through the proxy process. This imposes overhead on the proxy cache and may increase the end-to-end delay of the connections. To reduce this overhead, we propose to *splice* each pair of non-P2P connections using TCP splicing techniques [32], which have been used in layer-7 switching. For spliced connections, the sockets in the proxy process are closed and packets are relayed in the kernel stack instead of passing them up to the proxy process (in the application layer). Since packets do not traverse through the application layer, TCP splicing reduces overhead on maintaining a large number of open sockets as well as forwarding threads. Several implementation details had to be addressed. For example, since different connections start from different initial sequence numbers, the sequence numbers of packets over the spliced TCP connections need to be properly changed before being forwarded. This is done by keeping track of the sequence number difference between two spliced TCP connections, and updating sequence numbers before forwarding packets.

7 Experimental Evaluation of pCache

In this section, we conduct extensive experiments to evaluate all aspects of the proposed pCache system with *real* P2P traffic. We start our evaluation, in Section 7.1, by validating that our implementation of pCache is fully transparent, and it serves actual noncorrupted data to clients as well as saves bandwidth for ISPs. In Section 7.2, we show that pCache improves the performance of P2P clients without reducing the connectivity in P2P networks. In Section 7.3, we rigorously analyze the performance of the proposed storage management system and show that it outperforms others. In Section 7.4, we demonstrate that pCache is scalable and could easily serve most customer ASes in the Internet using a commodity PC. In Section 7.5, we validate the analysis of the inference algorithm and show that its estimates are correct in more than 99.7 percent of the cases with low overhead. Finally, in Section 7.6, we show that the proposed connection splicing scheme improves the performance of the cache.

The setup of the testbed used in the experiments consists of two separate IP subnets. Traffic from client machines in

subnet 1 goes through a Linux router on which we install pCache. Client machines in subnet 2 are directly attached to the campus network. All internal links in the testbed have 100 Mb/s bandwidth. All machines are configured with static IP addresses, and appropriate route entries are added in the campus gateway router in order to forward traffic to subnet 1. Our university normally filters traffic from some P2P applications and shapes traffic from others. Machines in our subnets were allowed to bypass these filters and shapers. pCache is running on a machine with an Intel Core 2 Duo 1.86 GHz processor, 1 GB RAM, and two hard drives: one for the operating system and another for the storage system of the cache. The operating system is Linux 2.6. In our experiments, we concurrently run 10 P2P software clients in each subnet. We could not deploy more clients because of the excessive volume of traffic they generate, which is problematic in a university setting (during our experiments, we were notified by the network administrator that our BitTorrent clients exchanged more than 300 GB in one day).

7.1 Validation of pCache and ISP Benefits

pCache is a fairly complex software system with many components. The experiments in this section are designed to show that the *whole* system actually works. This verification illustrates that:

1. P2P connections are identified and transparently split.
2. non-P2P connections are tunneled through the cache and are not affected by its existence.
3. segment lookup, buffering, storing, and merging all work properly and do not corrupt data.
4. the piece length inference algorithm yields correct estimates.
5. data are successfully obtained from external peers, assembled with locally cached data, and then served to internal peers in the appropriate message format.

All of these are shown for real BitTorrent traffic with many objects that have different popularities. The experiments also compare the performance of P2P clients with and without the proxy cache.

We modified an open source BitTorrent client called CTorrent. CTorrent is a lightweight (command-line) C++ program; we compiled it on both Linux and Windows. We did *not* configure CTorrent to contact or be aware of the cache in anyway. We deployed 20 instances of the modified CTorrent client on the four client machines in the testbed. Ten clients (in subnet 1) were behind the proxy cache and 10 others were directly connected to the campus network. All clients were controlled by a script to coordinate the download of many objects. The objects and the number of downloads per object were carefully chosen to reflect the actual relative popularity in BitTorrent as follows. We developed a crawler to contact popular torrent search engines such as TorrentSpy, MiniNova, and IsoHunt. We collected numerous torrent files. Each torrent file contains information about one object; an object can be a single file or multiple files grouped together. We also contacted trackers to collect the total number of downloads that each object received, which indicates the popularity of the object. We randomly took 500 sample objects from all the torrent files collected by our crawler, which were downloaded 111,620 times in total. The size distribution of the chosen 500 objects ranges from several hundred kilobytes to a few hundred megabytes. To conduct experiments within a reasonable amount of time, we scheduled about 2,700 download sessions. The number of download sessions assigned to an object is proportional to its popularity. Fig. 5 shows the distribution of the scheduled download sessions per objects. This figure indicates that our empirical popularity data follow Mandelbrot-Zipf distribution, which is a generalized form of Zipf-like distributions with an extra parameter to capture the flattened head nature of the popularity distribution observed near the most popular objects in our popularity data. The popularity distribution collect by our crawler is similar to those observed in previous measurement studies [1], [8].

After distributing the 2,700 scheduled downloads among the 500 objects, we randomly shuffled them such that downloads for the same object do not come back-to-back with each other, but rather they are interleaved with downloads for other objects. We then equally divided the 2,700 downloads into 10 lists. Each of these 10 lists, along with the corresponding torrent files, was given to a pair of CTorrent clients to execute. one in subnet 1 (behind the cache) and another in subnet 2. To conduct fair comparisons between the performance of clients with and without the cache, we made each pair of clients start a scheduled download session at the same time and with the same information. Specifically, only one of the two clients contacted the BitTorrent tracker to obtain the addresses of seeders and leechers of the object that need to be downloaded. This information was shared with the other client. Also, a new download session was not started unless the other client either finished or aborted its current session. We let the 20 clients run for *two days*, collecting detailed statistics from each client every 20 seconds.

Several sanity checks were performed and passed. First, all downloaded objects passed the BitTorrent checksum test. Second, the total number of download sessions that completed was almost the same for the clients with and without the cache. Third, clients behind the cache were regular desktops participating in the campus network, and other non-P2P network applications were running on them. These applications included web browsers, user authentication modules using centralized database (LDAP), and file system backup applications. All applications worked fine through the cache. Finally, a significant portion of the total traffic was served from the cache; up to 90 percent. This means that the cache identified, stored, and served P2P traffic. This is all happened *transparently* without changing the P2P protocol or configuring the client software. Also, since this is BitTorrent traffic, the piece length inference algorithm behaved as expected, and our cache did not interfere with the incentive scheme of BitTorrent. We note that the unusually high 90 percent byte hit rate seen in our experiments is due to the limited number of clients that we have; the clients did not actually generate enough traffic to fill the whole storage capacity of the cache. In full-scale deployment of the cache, the byte hit rate is expected to be smaller, in the range of 30 to 60 percent [11], which would still yield significant savings in bandwidth given the huge volume of P2P traffic.

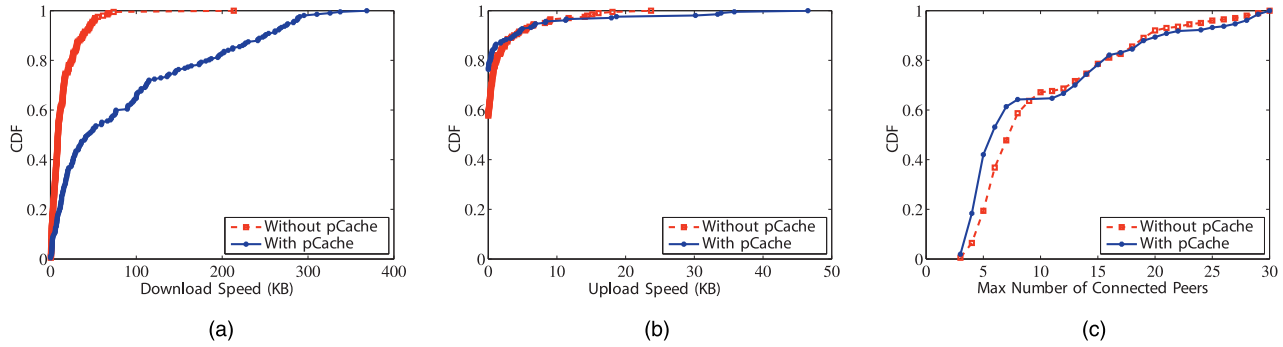

Fig. 4. The impact of the cache on the performance of clients. (a) Download speed. (b) Upload speed. (c) Number of connected peers.

7.2 Performance of P2P Clients

Next, we analyze the performance of the clients and the impact on the P2P network. We are interested in the download speed, upload speed, and number of peers to which each of our clients is connected. These statistics were collected every 20 seconds from all 20 clients. The download (upload) speed was measured as the number of bytes downloaded (uploaded) during the past period divided by the length of that period. Then, the average download (upload) speed for each completed session was computed. For the number of connected peers, we took the maximum number of peers that each of our clients was able to connect to during the sessions. We then used the maximum among all clients (in the same subnet) to compare the connectivity of clients with and without the cache. The results are summarized in Fig. 4. Fig. 4a shows that clients behind the cache achieved much higher download speed than other clients. This is expected as a portion of the traffic comes from the cache. Thus, the cache would benefit the P2P clients, in addition to benefiting the ISP deploying the cache by saving WAN traffic. Furthermore, the increased download speed did not require clients behind the cache to significantly increase their upload speeds, as shown in Fig. 4b. This is also beneficial for both clients and the ISP deploying the cache, while it does not hurt the global BitTorrent network, as clients behind the cache are still uploading to other external peers. The difference between the upload and download speeds can be attributed mostly to the contributions of the cache. Fig. 4c demonstrates that the presence of the cache did not reduce the connectivity of clients behind the cache; they have roughly the same number of connected peers. This is important for the local clients as well as the whole network, because reduced connectivity could lead to decreased availability of peers and the content stored on them. Finally, we notice that the cache may add some latency in the beginning of the download session. This latency was in the order of milliseconds in our experiments. This small latency is really negligible in P2P systems in which sessions last for minutes if not hours.

7.3 Performance of the Storage System

7.3.1 Experimental Setup

We compare the performance of the proposed storage management system against the storage system used in the widely deployed Squid proxy cache [31], after modifying it to serve partial hits needed for P2P traffic. This is done to answer the question: "What happens if we were to use the already-deployed web caches for P2P traffic?" To the best of our knowledge, we are not aware of any other storage systems designed for P2P proxy caches, and as described in Section 2, storage systems proposed in [21], [22], [23], [24] for web proxy caches and in [25], [26], [27] for multimedia caches are not readily applicable to P2P proxy caches. Therefore, we could not conduct a meaningful comparison with them.

We implemented the Squid system, which organizes files into a two-level directory structure: 16 directories in the first level and 256 subdirectories in every first-level directory. This is done to reduce the number of files in each directory in order to accelerate the lookup process and to reduce the number of i-nodes touched during the lookup. We implemented two versions of our proposed storage system, which are denoted by pCache/Fs and pCache/Raw in the plots. pCache/Fs is implemented on top of the ext2 Linux file system by opening a large file that is never closed. To write a segment, we move the file pointer to the appropriate offset and then write that segment to the file. The offset is determined by our disk block allocation algorithm. Reading a segment is done in a similar way. The actual writing/reading to the disk is performed by the ext2 file system, which might perform disk buffering and prefetching. pCache/Raw is implemented on a raw partition and it has complete control of reading/writing blocks from/to the disk. pCache/Raw uses direct disk operations.

In addition to the Squid storage system, we also implemented a storage system that uses a multidirectory structure, where segments of the same object are grouped together under one directory. We denote this storage

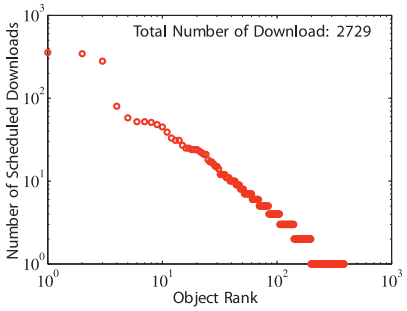

Fig. 5. Download sessions per object.

system as Multi-dir. This is similar to the previous proposals on clustering correlated web objects together for better disk performance, such as in [21]. We extended Multi-dir for P2P traffic to maintain the correlation among segments of the same object. Finally, we also implemented the Cyclic Object Storage System (COSS) which has recently been proposed to improve the performance of the Squid proxy cache [33, Section 8.4]. COSS sequentially stores web objects in a large file, and wraps around when it reaches the end of the file. COSS overwrites the oldest objects once the disk space is used up.

We isolate the storage component from the rest of the pCache system. All other components (including the traffic identification, transparent proxy, and connection splicing modules) are disabled to avoid any interactions with them. We also avoid interactions with the underlying operating system by installing two hard drives: one for the operating system and the other is dedicated to the storage system of pCache. No processes, other than pCache, have access to the second hard drive. The capacity of the second hard drive is 230 GB, and it is formatted into 4 KB blocks. The replacement policy used in the experiments is segment-based LRU, where the least-recently used segment is evicted from the cache.

We subject a specific storage system (e.g., pCache/Fs) to a long trace of 1.5 million segment requests; the trace collection and processing are described below. During the execution of the trace, we periodically collect statistics on the low-level operations performed on the disk. These statistics include the total number of: read operations, write operations, and head movements (seek length in sectors). We also measure the trace completion time, which is the total time it takes the storage system to process all requests in the trace. These low-level disk statistics are collected using the `blktrace` tool, which is an optional module in Linux kernels. Then, the whole process is repeated for a different storage system, but with the same trace of segment requests. During these experiments, we fix the total size of memory buffers at 0.5 GB. In addition, because they are built on top of the ext2 file system, Squid and pCache/Fs have access to the disk cache internally maintained by the operating system. Due to the intensive I/O nature of these experiments, Linux may substantially increase the size of the disk cache by stealing memory from other parts of the pCache system, which can degrade the performance of the whole system. To solve this problem, we modified the Linux kernel to limit the size of the disk cache to a maximum of 0.5 GB.

7.3.2 P2P Traffic Traces

We needed to stress the storage system with a large number of requests. We also needed the stream of requests to be reproducible such that comparisons among different storage systems are fair. To satisfy these two requirements, we collected traces from an operational P2P network instead of just generating synthetic traces. Notice that running the cache with real P2P clients (as in the previous section) would not give us enough traffic to stress the storage system, nor would it create identical situations across repeated experiments with different storage systems because of the high dynamics in P2P systems. To collect this trace, we modified an open source Gnutella client to run in superpeer mode and to simultaneously connect to up to 500 other superpeers in the network (the default number of connections is up to only 16). Gnutella was chosen because it has a two-tier structure, where ordinary peers connect to superpeers and queries/replies are forwarded among superpeers for several hops. This enabled us to passively monitor the network without injecting traffic. Our monitoring superpeer ran continuously for several months, and because of its high connectivity it was able to record a large portion of the query and reply messages exchanged in the Gnutella network. It observed query/reply messages in thousands of ASes across the globe, accounting to more than 6,000 tera bytes of P2P traffic. We processed the collected data to separate object requests coming from individual ASes. We used the IP addresses of the receiving peers and an IP-to-AS mapping tool in this separation. We chose one large AS (AS 9,406) with a significant amount of P2P traffic. The created trace contains: time stamp of the request, ID of the requested object, size of the object, and the IP address of the receiver. Some of these traces were used in our previous work [11], and are available online [14].

The trace collected from Gnutella provides realistic object sizes, relative popularities, and temporal correlation among requests in the same AS. However, it has a limitation: information about how objects are segmented and when exactly each segment is requested is not known. We could not obtain this information because it is held by the communicating peers and transferred directly between them without going through superpeers. To mitigate this limitation, we divided objects into segments with typical sizes, which we can know either from the protocol specifications or from analyzing a small sample of files. In addition, we generated the request times for segments as follows. The time stamp in the trace marks the start of downloading an object. A completion time for this object is randomly generated to represent peers with different network connections and the dynamic conditions of the P2P network. The completion time can range from minutes to hours. Then, the download time of each segment of the object is randomly scheduled between the start and end times of downloading that object. This random scheduling is not unrealistic, because this is actually what is being done in common P2P systems such as BitTorrent and Gnutella. Notice that using this random scheduling, requests for segments from different objects will be interleaved, which is also realistic. We sort all requests for segments based on their scheduled times and take the first 1.6 million requests to evaluate the storage systems. With this number of requests, some of our experiments took *two days* to finish.

7.3.3 Main Results

Some of our results are presented in Fig. 6 for a segment size of 0.5 MB. Results for other segment sizes are similar. We notice that the disk I/O operations resulted by the COSS storage system are not reported in Figs. 6a and 6b. This is because COSS storage system has a built-in replacement policy that leads to fewer cache hits, thus fewer read operations and many more write operations, than other storage systems. Since the COSS storage system results in different disk access pattern than that of other storage

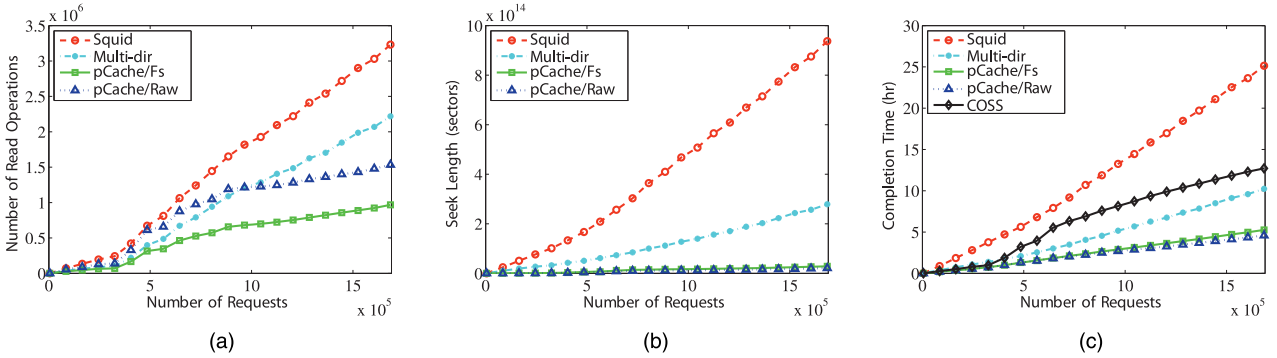

Fig. 6. Comparing the performance of the proposed storage management system on top of a raw disk partition (pCache/Raw) and on top of a large file (pCache/Fs) versus the Squid and other storage systems. (a) Read operations. (b) Head movements. (c) Completion time.

systems, we cannot conduct a meaningful low-level comparison between them in Figs. 6a and 6b. We will discuss more about the replacement policy of the COSS storage system in a moment. Fig. 6 demonstrates that the proposed storage system is much more efficient in handling the P2P traffic than other storage systems, including Squid, Multi-dir, and COSS. The efficiency is apparent in all aspects of the disk I/O operations: The proposed system issues a smaller number of read and write operations and it requires a much smaller number of disk head movements. Because of the efficiency in each element, the total time required to complete the whole trace (Fig. 6c) under the proposed system is less than five hours, while it is 25 hours under the Squid storage system. That is, the average service time per request using pCache/Fs or pCache/Raw is almost 1/5th of that time using Squid. This experiment also shows that COSS and Multi-dir improves the performance of the Squid system, but they have an inferior performance compared to our pCache storage system. Notice also that, unlike the case for Squid, Multi-dir, and COSS, the average time per request of our proposed storage system does not rapidly increase with number of requests. Therefore, the proposed storage system can support more concurrent requests, and it is more scalable than other storage systems, including the widely deployed Squid storage system.

Because it is optimized for web traffic, Squid anticipates a large number of small web objects to be stored in the cache, which justifies the creation of many subdirectories to reduce the search time. Objects in P2P systems, on the other hand, have various sizes and can be much larger. Thus, the cache could store a smaller number of P2P objects. This means that maintaining many subdirectories may actually add more overhead to the storage system. In addition, as Fig. 6b shows, Squid requires a larger number of head movements compared to our storage system. This is because Squid uses a hash function to determine the subdirectory of each segment, which destroys the locality among segments of the same object. This leads to many head jumps between subdirectory to serve segments of the same object. In contrast, our storage system performs segment merging in order to store segments of the same object near to each other on the disk, which reduces the number of head movements.

Fig. 6b also reveals that Multi-dir can reduce the number of head movements compared to the Squid storage system. This is because the Multi-dir storage system exploits the correlation among segments of the same P2P object by storing these segments in one directory. This segment placement structure enables the underlying file system to cluster correlated segments closer as most file systems cluster files in the same subdirectory together. Nevertheless, the overhead of creating many files/directories and maintaining their i-nodes was nontrivial. This can be observed in Fig. 6b where the average number of read operations of Multi-dir is slightly smaller than that of pCache/Raw when the number of cached objects is smaller (in the warm-up period), but rapidly increases once more objects are cached. We note that the Multi-dir results in fewer read operations than pCache/Raw in the warm-up period, because of the Linux disk buffer and prefetch mechanism. This advantage of Multi-dir quickly diminishes when the number of i-nodes increases.

We observe that COSS performs better than Squid but worse than Multi-dir in Fig. 6c. The main cause of its bad performance is because COSS uses the large file to mimic an LRU queue for object replacement, which requires it to write a segment to the disk whenever there is a cache hit. This not only increases the number of write operations but also reduces the effective disk utilization, because a segment may be stored on the disk several times although only one of these copies (the most recent one) is accessible. We plot the effective disk utilization of all considered storage systems in Fig. 7, where storage systems except COSS employ a segment-based LRU with high/low watermarks at 90 and 80 percent, respectively. In this figure, we

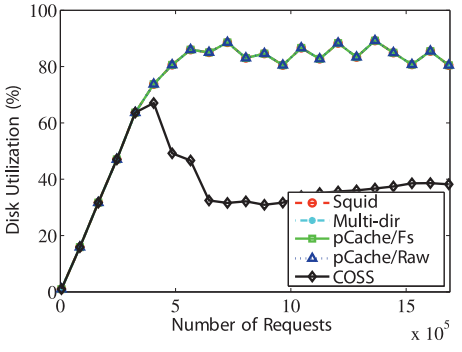

Fig. 7. Disk utilization for various storage systems.

observe that the disk utilization for COSS is less than 50 percent of that for other storage systems. Since COSS tightly couples the object replacement policy with the storage system, it is not desirable for P2P proxy caches, because their performance may benefit from replacement policies that capitalize on the unique characteristics of P2P traffic, such as the ones observed in [6], [8].

7.3.4 Additional Results and Comments

We analyzed the performance of pCache/Fs and pCache/Raw, and compared them against each other. As mentioned before, prefetching by the Linux file system may negatively impact the performance of pCache/Fs. We verified this by disabling this pre-fetching using the `hdparm` utility. By comparing pCache/Fs versus pCache/Raw using variable segment sizes, we found that pCache/Fs outperforms pCache/Raw when the segment sizes are small (16 KB or smaller). pCache/Raw, however, is more efficient for larger segment sizes. This is because when segment sizes are small, the buffering performed by the Linux file system helps pCache/Fs by grouping several small read/write operations together. pCache/Raw bypasses this buffering and uses direct disk operations. The benefit of buffering diminishes as the segment size increases. Therefore, we recommend using pCache/Fs if pCache will mostly serve P2P traffic with small segments such as BitTorrent. If the traffic is dominated by larger segments, as in the case of Gnutella, pCache/Raw is recommended.

7.4 Scalability of pCache

To analyze the scalability of pCache, ideally we should deploy thousands of P2P clients and configure them to incrementally request objects from different P2P networks. While this would test the whole system, it was not possible to conduct in our university setting, because it would have consumed too much bandwidth. Another less ideal option is to create many clients and connect them in *local* P2P networks. However, emulating the high dynamic behavior of realistic P2P networks is not straightforward, and will make our results questionable no matter what we do. Instead, we focus on the scalability of the slowest part, *the bottleneck*, of pCache, which is the storage system according to previous works in the literature such as [34]. All other components of pCache perform simple operations on data stored in the main memory, which is orders of magnitude faster than the disk. We acknowledge that this is only a partial scalability test, but we believe it is fairly representative.

We use the large trace of 1.5 million requests used in the previous section. We run the cache for about seven hours, and we stress the storage system by continuously submitting requests. We measure the average throughput (in Mbps) of the storage system every eight minutes, and we plot the average results in Fig. 8. We ignore the first one hour (warm-up period), because the cache was empty and few data swapping operations between memory and disk occur. The figure shows that in the steady state, an average throughput of more than 300 Mbps can easily be provided by our cache running on a *commodity PC*. This kind of throughput is probably more than enough for the majority of customer ASes such as universities and small ISPs, because the total capacities of their Internet access links are typically smaller than the maximum throughput that can be achieved by pCache. For large ISPs with Gbps links, a high-end server with high-speed disk array could be used. Notice that our pCache code is not highly optimized for performance. Notice also that the 300 Mbps is the throughput for P2P traffic only, not the whole Internet traffic, and it represents a worst-case performance because the disk is continuously stressed.

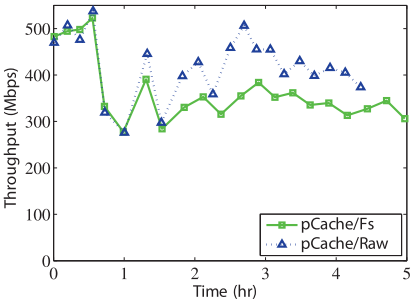

Fig. 8. Throughput achieved by pCache.

7.5 Evaluation of the Inference Algorithm

We empirically evaluate the algorithm proposed in Section 4 for inferring the piece length in BitTorrent traffic. We deployed 10 CTorrent clients on the machines behind the cache. Each client was given a few thousand torrent files. For each torrent file, the client contacted a BitTorrent tracker to get potential peers. The client reconnected to the tracker if the number of peers dropped below 10 to obtain more peers. After knowing the peers, the client started issuing requests and receiving traffic. The cache logged all requests during all download sessions. Many of the sessions did not have any traffic exchanged, because there were not enough active peers trading pieces of those objects anymore, which is normal in BitTorrent. These sessions were dropped after a time-out period. We ran the experiment for several days. The cache collected information from more than 2,100 download sessions. We applied the inference algorithm on the logs and we compared the estimated piece lengths against the actual ones (known from the torrent files).

The results of this experiment are presented in Fig. 9. The x-axis shows the number of samples taken to infer the piece length, and the y-axis shows the corresponding accuracy achieved. The accuracy is computed as the number of correct inferences over the total number of estimations, which is 2,100. We plot the theoretical results computed from (1) and (2) and the actual results achieved by our algorithm with and without the improvement heuristic. The figure shows that the performance of our basic algorithm using actual traffic is very close to the theoretical results, which validates our analysis in Section 4. In addition, the simple heuristic improved the inference algorithm significantly. The improved algorithm infers the piece length correctly in about 99.7 percent of the cases, which is done by using only six samples on average. From further analysis, we found that the samples used by the algorithm correspond to less than three percent of each object on average, which shows how fast the inference is done in terms of the object's size. Keeping this portion small is

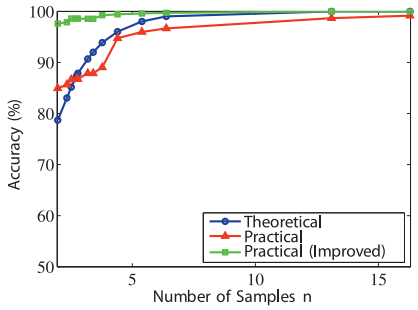

Fig. 9. Accuracy of the inference algorithm.

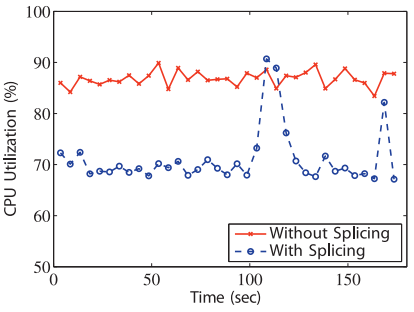

Fig. 10. CPU load reduction due to connection splicing.

important, because the cache does not start storing segments of an object until it knows its piece length. This is done to simplify the implementation of the cache; an alternative is to put the observed samples in a temporary storage till the piece length is inferred.

7.6 Performance of Connection Manager

Finally, we evaluate the performance gain from the connection splicing technique, which is designed to tunnel non-P2P traffic through the cache without overloading it. To fully stress our pCache, we use traffic generators to create many TCP connections through the cache, where each traffic generator sends as fast as possible. We vary the number of traffic generators. We measure the load on the cache in terms of memory usage and CPU utilization with and without connection splicing. Our logs show a reduction in the number of threads created to manage connections and the memory used. The number of threads is reduced because upon splicing two TCP connections together, the kernel closes the local TCP sockets and directly forwards packets inside the kernel space, which relieves pCache from keeping two forwarding threads. We plot in Fig. 10, a sample CPU utilization of 64 traffic generators with and without connection splicing. The figure shows that splicing reduces the CPU utilization by at least 10 percent. Furthermore, our experiments show that, without connection splicing, the CPU load increases when the number of traffic generators increases. However, with connection splicing, the CPU load is rather constant.

8 Conclusions and Future Work

It has been demonstrated in the literature that objects in P2P systems are mostly immutable and the traffic is highly repetitive. These characteristics imply that there is a great potential for caching P2P traffic to save WAN bandwidth and to reduce the load on the backbone links. To achieve this potential, in this paper, we presented pCache, a proxy cache system explicitly designed and optimized to store and serve P2P traffic from different P2P systems. pCache is fully transparent and it does not require any modifications to the P2P protocols. Therefore, it could be readily deployed by ISPs and university campuses to mitigate some of the negative effects of the enormous amount of P2P traffic. pCache has a modular design with well-defined interfaces, which enables it to support multiple P2P systems and to easily accommodate the dynamic and evolving nature of P2P systems. Using our prototype implementation of pCache in our campus network, we validated the correctness of our design. While designing and implementing pCache, we have identified and justified all key issues relevant to developing proxy caches for P2P traffic. These include customized storage system, transparent handling of P2P connections, efficient tunneling of non-P2P connections through the cache, and inferring required information for caching and serving requests.

We proposed a new storage management system for proxy caches of P2P traffic. This storage system supports serving requests for arbitrary byte ranges of stored objects—a requirement in P2P systems. We compared the proposed storage system against other storage systems, including the one in the widely deployed Squid proxy cache. Our comparison showed that the average service time per request using our storage system is almost 1/5th of that time using Squid. Our storage system also outperforms the Multi-dir storage system that utilizes correlation among segments and the COSS storage system which is a recent file system proposed to improve the performance of Squid. In addition, we proposed and evaluated an algorithm to estimate the piece length of different objects in BitTorrent. This information is required by the cache, but it is not included in the messages exchanged between peers.

We are currently working on several extensions for pCache. One of them is to handle encrypted P2P traffic. We are also working on designing new object replacement policies to be used with pCache. Last, we are exploring the potential of cross-system caching. This means that if pCache stores an object downloaded from one P2P system, it can serve requests for that object in another P2P system.

Acknowledgments

This work is partially supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

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," *Proc. ACM Symp. Operating Systems Principles (SOSP '03)*, pp. 314-329, Oct. 2003.

[2] S. Sen and J. Wang, "Analyzing Peer-to-Peer Traffic across Large Networks," *IEEE/ACM Trans. Networking*, vol. 12, no. 2, pp. 219-232, Apr. 2004.

[3] T. Karagiannis, A. Broido, N. Brownlee, K.C. Claffy, and M. Faloutsos, "Is P2P Dying or Just Hiding?" *Proc. IEEE Global Telecomm. Conf. (GLOBECOM '04)*, pp. 1532-1538, Nov. 2004.

[4] T. Karagiannis, P. Rodriguez, and K. Papagiannaki, "Should Internet Service Providers Fear Peer-Assisted Content Distribution?" *Proc. ACM Conf. Internet Measurement (IMC '05)*, pp. 63-76, Oct. 2005.
[5] R. Bindal, P. Cao, W. Chan, J. Medved, G. Suwala, T. Bates, and A. Zhang, "Improving Traffic Locality in BitTorrent via Biased Neighbor Selection," *Proc. IEEE Int'l Conf. Distributed Computing Systems (ICDCS '06)*, pp. 66-74, July 2006.
[6] A. Wierzbicki, N. Leibowitz, M. Ripeanu, and R. Wozniak, "Cache Replacement Policies Revisited: The Case of P2P Traffic," *Proc. Int'l Workshop Global and Peer-to-Peer Computing (GP2P '04)*, pp. 182-189, Apr. 2004.
[7] G. Shen, Y. Wang, Y. Xiong, B. Zhao, and Z. Zhang, "HPTP: Relieving the Tension between ISPs and P2P," *Proc. Int'l Workshop Peer-to-Peer Systems (IPTPS '07)*, Feb. 2007.
[8] O. Saleh and M. Hefeeda, "Modeling and Caching of Peer-to-Peer Traffic," *Proc. IEEE Int'l Conf. Network Protocols (ICNP '06)*, pp. 249-258, Nov. 2006.
[9] N. Leibowitz, A. Bergman, R. Ben-Shaul, and A. Shavit, "Are File Swapping Networks Cacheable?" *Proc. Int'l Workshop Web Content Caching and Distribution (WCW '02)*, Aug. 2002.
[10] D. Stutzbach, S. Zhao, and R. Rejaie, "Characterizing Files in the Modern Gnutella Network," *Multimedia Systems*, vol. 13, no. 1, pp. 35-50, Sept. 2007.
[11] M. Hefeeda and O. Saleh, "Traffic Modeling and Proportional Partial Caching for Peer-to-Peer Systems," *IEEE/ACM Trans. Networking*, vol. 16, no. 6, pp. 1447-1460, Dec. 2008.
[12] Ipoque Internet Study, http://www.ipoque.com/resources/internet-studies/internet-study-2008_2009, 2009.
[13] M. Hefeeda, C. Hsu, and K. Mokhtarian, "pCache: A Proxy Cache for Peer-to-Peer Traffic," *Proc. ACM SIGCOMM '08*, pp. 995-996, Aug. 2008.
[14] Network Systems Lab, http://nsl.cs.sfu.ca/wiki/, 2011.
[15] S. Sen, O. Spatscheck, and D. Wang, "Accurate, Scalable In-Network Identification of P2P Traffic Using Application Signatures," *Proc. Int'l World Wide Web Conf. (WWW '04)*, pp. 512-521, May 2004.
[16] T. Karagiannis, A. Broido, M. Faloutsos, and K. Claffy, "Transport Layer Identification of P2P Traffic," *Proc. ACM Conf. Internet Measurement (IMC '04)*, pp. 121-134, Oct. 2004.
[17] A. Spognardi, A. Lucarelli, and R. Di Pietro, "A Methodology for P2P File-Sharing Traffic Detection," *Proc. Int'l Workshop Hot Topics in Peer-to-Peer Systems (HOT-P2P '05)*, pp. 52-61, July 2005.
[18] A. Madhukar and C. Williamson, "A Longitudinal Study of P2P Traffic Classification," *Proc. IEEE Int'l Symp. Modeling, Analysis, and Simulation of Computer and Telecomm. Systems (MASCOTS '06)*, pp. 179-188, Sept. 2006.
[19] OverCache MSP Home Page, http://www.oversi.com/products/overcache-msp, 2011.
[20] PeerApp UltraBand Home Page, http://www.peerapp.com/products-ultraband.aspx, 2009.
[21] A. Abhari, S. Dandamudi, and S. Majumdar, "Web Object-Based Storage Management in Proxy Caches," *Future Generation Computer Systems*, vol. 22, no. 1, pp. 16-31, Jan. 2006.
[22] E. Shriver, E. Gabber, L. Huang, and C. Stein, "Storage Management for Web Proxies," *Proc. USENIX Ann. Technical Conf. (USENIX '01)*, pp. 203-216, June 2001.
[23] J. Wang, R. Min, Y. Zhu, and Y. Hu, "UCFS—A Novel User-Space, High Performance, Customized File System for Web Proxy Servers," *IEEE Trans. Computers*, vol. 51, no. 9, pp. 1056-1073, Sept. 2002.
[24] E. Markatos, D. Pnevmatikatos, M. Flouris, and M. Katevenis, "Web-Conscious Storage Management for Web Proxies," *IEEE/ACM Trans. Networking*, vol. 10, no. 6, pp. 735-748, Dec. 2002.
[25] J. Liu and J. Xu, "Proxy Caching for Media Streaming over the Internet," *IEEE Comm. Magazine*, vol. 42, no. 8, pp. 88-94, Aug. 2004.
[26] K. Wu, P. Yu, and J. Wolf, "Segmentation of Multimedia Streams for Proxy Caching," *IEEE Trans. Multimedia*, vol. 6, no. 5, pp. 770-780, Oct. 2004.
[27] S. Chen, B. Shen, S. Wee, and X. Zhang, "SProxy: A Caching Infrastructure to Support Internet Streaming," *IEEE Trans. Multimedia*, vol. 9, no. 5, pp. 1062-1072, Aug. 2007.
[28] J. Ledlie, "Damelo! An Explicitly Co-Locating Web Cache File System," master's thesis, Dept. of Computer Science, Univ. of Wisconsin, Dec. 2000.
[29] P4P Working Group, http://www.openp4p.net/, 2009.
[30] H. Xie, Y. Yang, A. Krishnamurthy, Y. Liu, and A. Silberschatz, "P4P: Portal for (P2P) Applications," *Proc. ACM SIGCOMM '08*, pp. 351-362, Aug. 2008.
[31] Squid Home Page, http://www.squid-cache.org/, 2011.
[32] TCPSP Home Page, http://www.linuxvirtualserver.org/software/tcpsp/index.html, 2011.
[33] D. Wessels, *Squid: The Definitive Guide*, first ed., O'Reilly, 2004.
[34] E. Markatos, M. Katevenis, D. Pnevmatikatos, and M. Flouris, "Secondary Storage Management for Web Proxies," *Proc. USENIX Symp. Internet Technologies and Systems (USITS '99)*, pp. 93-104, Oct. 1999.

Mohamed Hefeeda received the BSc and MSc degrees from Mansoura University, Egypt, in 1994 and 1997, respectively, and the PhD degree from Purdue University, West Lafayette, Indiana, in 2004. He is an associate professor in the School of Computing Science, Simon Fraser University, Surrey, British Columbia, Canada, where he leads the Network Systems Lab. His research interests include multimedia networking over wired and wireless networks, peer-to-peer systems, mobile multimedia, and Internet protocols. Dr. Hefeeda won the Best Paper Award at the IEEE Innovations 2008 conference for his paper on the hardness of optimally broadcasting multiple video streams with different bitrates. In addition to publications, he and his students have developed actual systems, such as pCache, svcAuth, pCDN, and mobile TV testbed. The mobile TV testbed software developed by his group won the Best Technical Demo Award at the ACM Multimedia 2008 conference. He serves as the Preservation Editor of the ACM Special Interest Group on Multimedia (SIGMM) web magazine. He served as the program chair of the ACM International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 2010) and as a program cochair of the International Conference on Multimedia and Expo (ICME 2011). In addition, he has served on many technical program committees of major conferences in his research areas, including ACM Multimedia, ACM Multimedia Systems, and the IEEE Conference on Network Protocols (ICNP). He is on the editorial boards of the *ACM Transactions on Multimedia Computing, Communications and Applications (ACM TOMCCAP)*, the *Journal of Multimedia*, and the *International Journal of Advanced Media and Communication*. He is a senior member of the IEEE.

Cheng-Hsin Hsu received the BSc and MSc degrees from the National Chung-Cheng University, Taiwan, in 1996 and 2000, respectively, the MEng degree from the University of Maryland, College Park, in 2003, and the PhD degree from Simon Fraser University, British Columbia, Canada, in 2009. He is a senior research scientist at Deutsche Telekom R&D Lab USA, Los Altos, California. His research interests are in the area of multimedia networking and distributed systems. He is a member of the IEEE.

Kianoosh Mokhtarian received the BSc degree in Software Engineering from Sharif University of Technology, Iran, in 2007 and the MSc degree in Computing Science from Simon Fraser University, British Columbia, Canada, in 2009. He worked as a software engineer in the networking industry at Mobidia Inc. and Fortinet Inc., British Columbia, Canada, and is currently a PhD student in Electrical and Computer Engineer at the University of Toronto. His research interests include peer-to-peer systems and multimedia networking. He is a student member of the IEEE.