Computer Networks 54 (2010) 1229–1241

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

Proxy caching for peer-to-peer live streaming ☆

Ke Xu [a], Ming Zhang [b,*], Jiangchuan Liu [c], Zhijing Qin [d], Mingjiang Ye [a]

[a] Department of Computer Science, Tsinghua University, Beijing 100084, PR China
[b] School of Software, Tsinghua University, Beijing 100084, PR China
[c] School of Computing Science, Simon Fraser University, Vancouver, BC, Canada
[d] School of Software and Microelectronics, Peking University, Beijing 100871, PR China

ARTICLE INFO

Article history:
Received 6 May 2009
Received in revised form 25 September 2009
Accepted 29 November 2009
Available online 6 December 2009
Responsible Editor: I. Habib

Keywords:
P2P
Live streaming
Proxy caching

ABSTRACT

Peer-to-Peer (P2P) live streaming has become increasingly popular over the Internet. To alleviate the inter-ISP traffic load and to minimize the access latency, proxy caching has been widely suggested for P2P applications. In this paper, we carry out an extensive measurement study on the properties of P2P live streaming data requests. Our measurement demonstrates that the P2P living streaming traffic exhibits strong localities that could be explored by caching. This is particularly noticeable for the temporal locality, which is often much weaker in the conventional P2P file sharing applications. Our results further suggest that the request time of the same data piece from different peers exhibits a generalized extreme value distribution. We then propose a novel sliding window (SLW)-based caching algorithm, which predicts and caches popular data pieces according to the measured distribution. Our experimental results suggest that the P2P live streaming can greatly benefit from the proxy caching. And, with much lower overhead, our SLW algorithm works closer to an off-line optimal algorithm that holds the complete knowledge of future requests.

1. Introduction

Recent years have witnessed the tremendous success of the peer-to-peer (P2P) communication paradigm. With each participating node contributing its own resources, the P2P communication architecture scales extremely well with user population. It has been widely used in such applications as file sharing [1,2], voice over IP (VoIP) [3], live streaming and video-on-demand (VOD) [4]. They together have contributed to a great portion of the overall Internet traffic [5], and its ever growing trend has posed a significant threat to sustainable operations of Internet Service Providers (ISP) [6].

To mitigate the traffic load, particularly the costly inter-ISP traffic, caching data of interest closer to end-users has been frequently suggested in the literature. There have been extensive studies on caching the traffic of web [7,8] or streaming video in the client/server architecture (video streaming for short) [9]. Recent works have also examined caching for P2P file sharing [6,10]. The caches are generally deployed at gateways of institutions, referred to as *proxy caching*. Through satisfying requests from the local storage, they not only reduces the bandwidth consumption, but also minimizes the access latency [11]. The latest experiments further suggests that caching is very effective for P2P file sharing, too (with a bandwidth reduction of over 60% [12]).

There are however important differences between living streaming and the conventional web and file sharing applications. For the latter two, the data pieces exhibit a skewed distribution, and the pieces frequently requested

☆ This work has been supported in part by NSFC Project (60970104, 60873253), NSFC-RGC Joint Research Project (20731160014), 973 Project of China (2009CB320501), 863 Project of China (2008AA01A326) and Program for New Century Excellent Talents in University.

* Corresponding author. Tel.: +86 6 260 3064; fax: +86 6 277 0753.
E-mail addresses: xuke@tsinghua.edu.cn (K. Xu), tigerCN7@gmail.com (M. Zhang), jcliu@cs.sfu.ca (J. Liu), qzj@csnet1.cs.tsinghua.edu.cn (Z. Qin), yemingjiang@csnet1.cs.tsinghua.edu.cn (M. Ye).

1389-1286/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.comnet.2009.11.013

in the past tend to be frequently requested in the future. This is not the case in living streaming that demands sequential playback. The data pieces in living streaming on the other hand has strong temporal correlations, since they are exchanged in a small playback window. With the playback window moves over time, the popularity and even usefulness of the data pieces change as well. To the best of our knowledge, proxy caching for P2P live streaming traffic has yet to be examined, despite several pioneer studies on collaborative caching on the peer side [13,14].

In this paper, we present a systematic study on the proxy caching for P2P live streaming. We first analyze the real data request of a popular P2P live streaming application, PPLive [4], and identify its key characteristics. In particular, we find that the request time of the same data piece from different peers exhibits a generalized extreme value distribution. We then develop a data request generator that can closely synthesize P2P live streaming traffic. We further propose a novel sliding window (SLW) caching algorithm that explores the unique distribution of the live streaming requests.

We evaluated the performance of our algorithm and compare it with typical caching strategies. Our experiments show that our algorithm well suites the P2P live streaming traffic, and its cache hit rate is very close to that of an off-line optimal algorithm. We further model the theoretical performance of the new algorithm, and provides general guidelines for the configuration and optimization of the algorithm.

The rest of the paper is organized as follows. In Section 2, we survey the related work. In Section 3, we compare the traffic characteristics of P2P live streaming and P2P file sharing, followed by a measurement study on the data request distribution of PPLive in Section 4. We then present our caching algorithm in Section 5, and evaluate its performance in Section 6. Section 6 further models the performance of our caching algorithm. Finally, Section 7 concludes the paper.

2. Related work

There have been a series of measurement studies on P2P live streaming systems, particularly PPLive [15,16]. Hei et al. [15] carry out an in-depth analysis of PPLive, and find that PPLive users experience large start-up delay and playback lags. They conclude that dedicated proxy nodes are necessary to help with delivering videos at higher playback rate. According to Ali et al. [16], P2P live streaming has a greater impact on network bandwidth than P2P file sharing, because the upload bandwidth is unfairly exhausted by some peers. These studies reveal a lot of statistical information about PPLive, but the request distribution, which is of critical importance to cache design, has yet to be identified.

In the past, web caching has been extensively studied [8]. The importance and feasibility of caching P2P traffic have been demonstrated in [17,12] and [18]. Studies [17] and [12] show that P2P traffic is highly redundant and caching can reduce as much as 50–60% of the traffic. The work in [18] further suggests deploying proxy caches and making P2P protocols locality-aware, so as to reduce inter-ISP traffic. Other studies on proxy caching for P2P file sharing include [6] and [10]. We summary the key issues addressed in these studies as follows:

- *Object popularity*. Incorporating object popularity into caching algorithms can help with improving the cache hit rate. The pattern of object popularity is usually studied as part of the caching algorithm design process [6]. Typical object popularity patterns can be found in [7] and [19]. These studies all prove the effectiveness of the Pareto principle (or 80–20 rule). The channel popularity in P2P live streaming also satisfies the 80–20 rule which is an important consideration for caching algorithm design in our work.
- *Temporal and spatial locality*. Locality of reference characterizes the ability to predict future accesses to objects from the history information [8]. There are two important types of locality: temporal and spatial. Temporal locality refers to the repeated accesses to the same object within short time periods. It implies that recently accessed objects are likely to be accessed again in the future. Spatial locality refers to the patterns that accesses to some objects imply accesses to certain other objects. That said, the references to some objects can be a predictor of future references to other objects. Least Recently Used (LRU) is a classic algorithm utilizing temporal locality. The SLW algorithm proposed in our work exploits both the temporal and spatial locality of data requests.

Despite the common issues, existing studies have suggested that web, video streaming and P2P file sharing traffic all have their distinct features, and specialized policies are preferred to cache their respective traffic [10]. Also note that the popular pre-fetching policies to minimize the startup latency or to meet certain time constraints [9,20–23] are not necessarily useful for living streaming. This is because the data pieces are released gradually over time. In this work, we concentrate on achieving best cache hit rate without considering time constraint explicitly. We will see later that these two goals actually do not conflict.

3. Proxy caching for P2P live streaming: a general view

In this section, we first give a generic architectural view of caching for P2P live streaming, in particular, where and how the caches are deployed. We then analyze the differences between live streaming and file sharing in terms of cache design, which motivates our study.

3.1. Generic cache architecture for P2P traffic

In the P2P communication paradigm, each node (peer) has dual roles: downloading from other peers as a client, and uploading to other peers as a server. To avoid long-distance sessions, particularly those between peers of different institutions such as Autonomous Systems (ASes), the cache servers can be deployed at the access points of regional networks (as in [24]), as shown in Fig. 1a.

Once operated, the gateway intercepts the P2P downloading requests originated from its associated regional network and redirects them to the P2P cache server

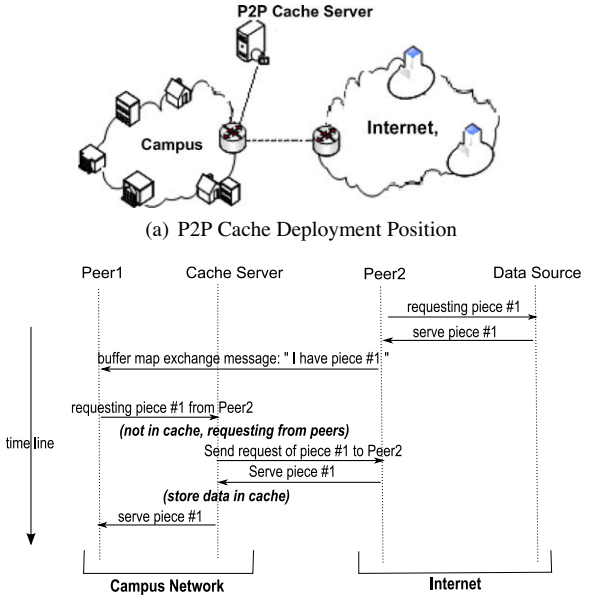


Fig. 1. P2P cache deployment.

(Fig. 1). Such interception can be either implemented by redirecting signals or using forged IP addresses of remote peers [25]. The objective is to serve as many data requests locally as possible. The server will fetch data from remote peers only when the data cannot be found in its local cache. The old data will be evicted if the cache space is exhausted. Note that the cache server is transparent to P2P peers, since only the downloading requests are intercepted while other control signals such as buffer map exchange messages remain untouched.

Fig. 1 shows that every downloading request sent to the Internet will first be processed by the cache server. Nevertheless, the peers in the same regional network may also exchange data, e.g., using the collaborative cache mechanism to fetch as much data locally as possible [18].

3.2. Distinct features of P2P live streaming

Numerous P2P live streaming applications have been developed, including CoolStreaming [26], PPLive [4] and PPStream [27]. The design of these applications shares many similarities with that of P2P file distribution, particularly BitTorrent [1]. They both divide the whole file into smaller data pieces and maintain a mesh-like overlay among the peers for them to exchange the data availability information, issue requests, and then fetch data from partners.

There are nevertheless important differences between P2P live streaming and P2P file sharing (Table 1). First, only the data pieces around the playback point are requested in live streaming, while any data piece may be exchanged at any time in file sharing. Second, given the skewed popularity, a frequently accessed file can be cached as a whole to save bandwidth; in live streaming, caching the whole object is generally impossible because the data is unavailable

Table 1
Comparison between P2P live streaming and P2P file sharing.

	P2P file sharing	P2P live streaming
Exchanged data	Can be any part of an object	Only those around the playback point
Caching unit	Whole file/whole segment	Only segments
Cached data	Can be popular for a long time	become obsolete soon
Popular data	Can be discrete parts of a file/the whole file	Continuous and always around the playback point

until being released. In general, the data requests generated by live streaming peers are semi-ordered and continuous, concentrating in a limited window, as shown in Fig. 2. As the playback window moves over time, the data popularity changes as well. In contrast, file sharing may have multiple hot spots that are relatively stable. Such differences imply that the existing caching algorithms of P2P file sharing must be substantially re-designed to fit the uniqueness of living streaming.

4. Data request analysis and data request synthesis for P2P live streaming

In P2P applications, peers exchange data by sending data requests to other peers. Existing studies on caching P2P file sharing traffic mainly focus on the spatial characteristics of data requests, including the amount of requests for each object and the request distribution among segments inside the object. In this section, we study the temporal characteristics of live streaming data requests with real data traces from PPLive.

PPLive is a typical P2P live streaming system which is very popular in China and has a significant user base in North America as well. It employs the mesh-pull P2P data exchange architecture, which is also used in CoolStreaming, PPStream, UUSee, SopCast, TVAnts, VVSky and many other P2P IPTV systems [15]. Because data pieces get stale soon in live streaming, it is always of high priority to fetch the rarest and the most urgent data pieces to meet the real-time constraints. Considering the applications all use the mesh-pull architecture, the request scheduling mechanisms are similar. Thus we believe PPLive is representative of these live streaming applications in the perspective of data request characteristics.

The traces are collected by passively sniffing the PPLive clients. When a user opens the PPLive software, the endpoint is activated by joining the overlay network. It re-

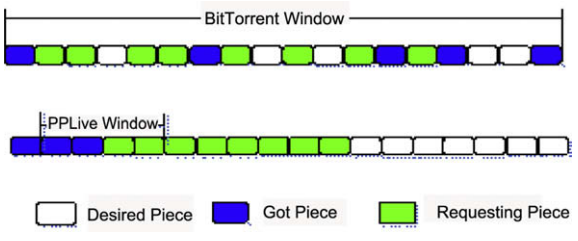


Fig. 2. Windows for BitTorrent and PPLive.

trieves channel list and peer list from root servers and exchanges buffer map with partners. Finally, it sends data requests to other peers for video data. These requests are collected with a proprietary tool provided by PPLive company (we have cooperation with PPLive) and stored for off-line analysis.

Typically the whole movie file is divided into *data segments* each containing sixteen *data pieces*. The information stored for a data request includes request sending time (also known as *time stamp*), request segment sequence and data piece sequence (Fig. 3). The data requests are sent based on data piece to different peers. The collected requests for each host are then preprocessed by filtering duplicate requests, removing segment sequence number and translating the relative data piece sequence number into absolute one. We have collected traces generated by five PPLive clients on different hosts in our campus watching the same program for 2 h.

4.1. Characteristics of data requests for individual users

In this section, we study the characteristics of data requests for single users.

4.1.1. Stable requesting rate

We plot the data requests as points (Fig. 4a) and find the linear relationship between the data piece sequence number and time stamp. It indicates that the *requesting rate* (requests sent per second) represented by the slope is stable. We have also studied requests for other channels and on different platforms. The linear relationship is also observed though the slope may be different.

4.1.2. Request groups and subgroups

We have found grouping features of data requests. In Fig. 4a, the requests in the box are sent in 1 s (e.g., from 101.5 s to 102.5 s). We call these requests a *request group*. Requests in a request group can be further divided into two subgroups, *request subgroup1* and *request subgroup2*, according to the thickness of request time stamp. The interval between the centers of the two subgroups is about 0.5 s. We define the *group/subgroup size* as the number of requests in a group/subgroup. From Fig. 4b and c, we learn that most groups are of size 32 or 48. And the later accounts for about 75% of all the groups. This implies that PPLive client is designed to send 32 or 48 data requests per second. To learn the proportion of requests sent in subgroup1 to those sent in the whole group, we plot the $\frac{subgroup1\ size}{group\ size}$ distribution for group size 32 and 48, respectively (Fig. 4e and f). Both figures show that most of the requests are sent in subgroup1, and relatively more requests are sent in subgroup 2 for group size 48. We have also studied the interval between adjacent requests of the same subgroup. Fig. 4d shows that more than 97% is between 0 and 0.0005 s.

These study results reveal the data request generation mechanisms for single users, which are the basis of the data request generator algorithm in Section 4.4.

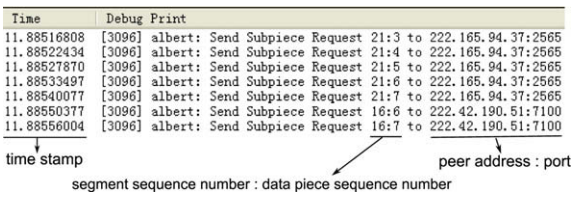


Fig. 3. Data request information.

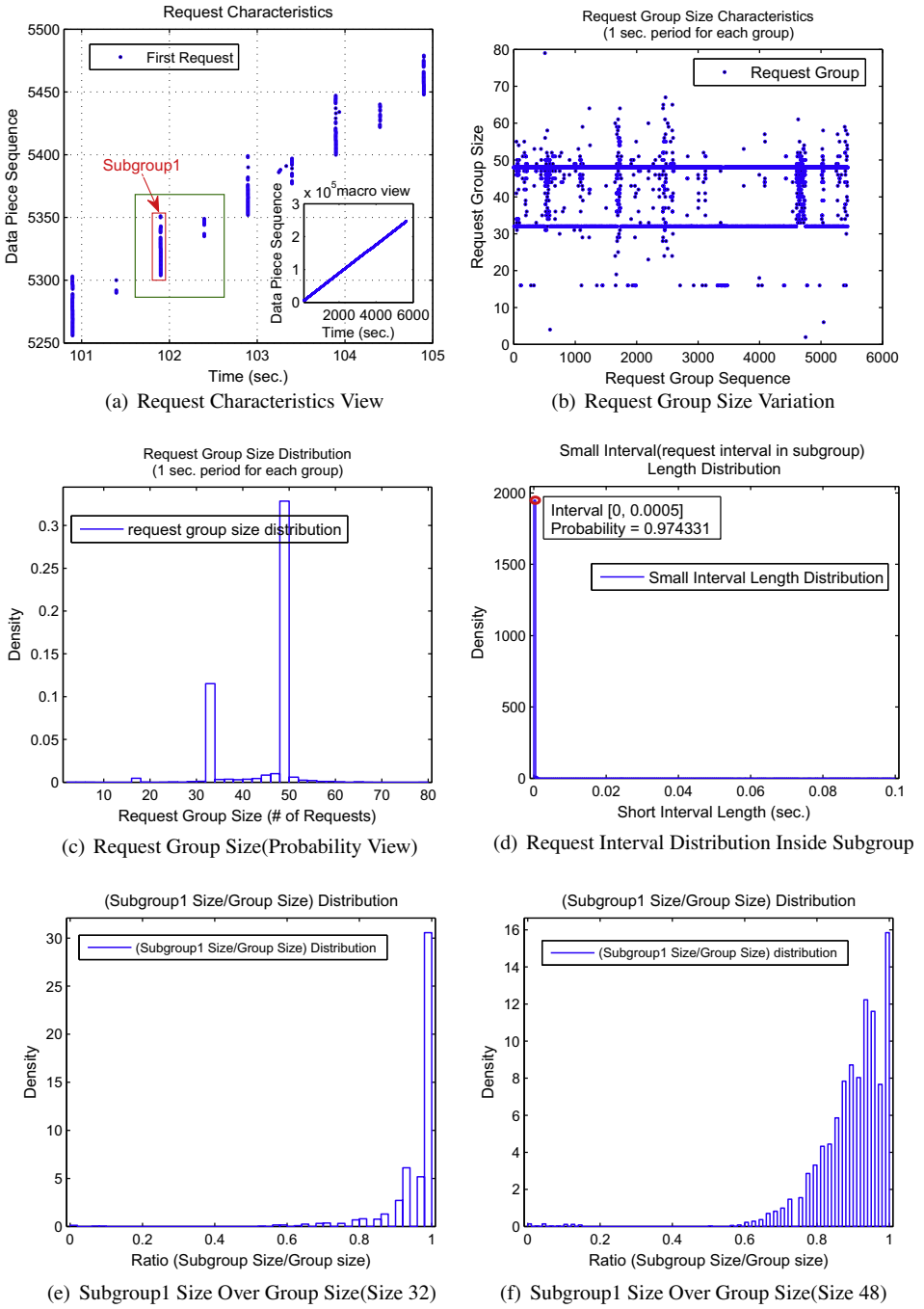
4.2. Request lag distribution among peers

Playback lag refers to the phenomenon that some peers watch frames in a channel minutes behind other peers [15]. Here, by *request lag*, we mean the phenomenon that some peers fetch a data piece in a channel seconds behind other peers in the same network. Request lag is very important because it indicates the lifetime of a data piece in live streaming. The interval between the first and last requests of a data piece is the *lag length*. If a data piece is already released and not yet obsolete (still being requested), we say it is *active*. The number of active data pieces in a channel at any given time is limited. These active data pieces are continuous, which are called the *requesting window* of a channel.

To study the request lag characteristics, we divide the data requests from five hosts into smaller segments each with 500 data requests, and 4140 request segments are acquired. These request segments are all normalized to start from request sequence number 0 and time stamp 0.0. Each request segment represents the requests of a single user. So we obtained requests of 4140 peers from the data captured in five clients. Then the lags among peers fetching the same data piece are analyzed. We retrieve the time stamp (logged in each request record) of data request number 300 (the 300th of 500 requests, which is around the middle) from each group, calculate the lag to the earliest request of the group and plot its probability distribution in Fig. 5. By fitting the distribution of request lags in matlab with distribution models including exponential, extreme value, generalized extreme value, generalized pareto, logistic, normal and t location-scale, we find that it follows the generalized extreme value (GEV) distribution with parameters $\kappa = 0.214242$, $\mu = 2.46523$, $\sigma = 1.992$ 42 (Fig. 5) and the log likelihood is -25582.2. The GEV distribution fits the data well in the view of both probability density and cumulative probability. Another way to measure the distribution of a pile of data is empirical distribution. It usually can fit the real data better. But the distribution function of empirical distribution can be complicated and not typical. We tend to fit the data with a well studied distribution and the benefits will be found when analyzing the performance of our new caching algorithm.

The GEV distribution is widely used in risk management, finance, insurance, economics, hydrology, material sciences, telecommunications, and many other industries dealing with extreme events [28]. Its probability density function (PDF) can be expressed as ($\kappa \neq 0$)

$$f(x) = \frac{1}{\sigma}\lambda^{-1-\frac{1}{\kappa}}e^{-\lambda^{-\frac{1}{\kappa}}} \tag{1}$$



(a) Request Characteristics View

(b) Request Group Size Variation

(c) Request Group Size(Probability View)

(d) Request Interval Distribution Inside Subgroup

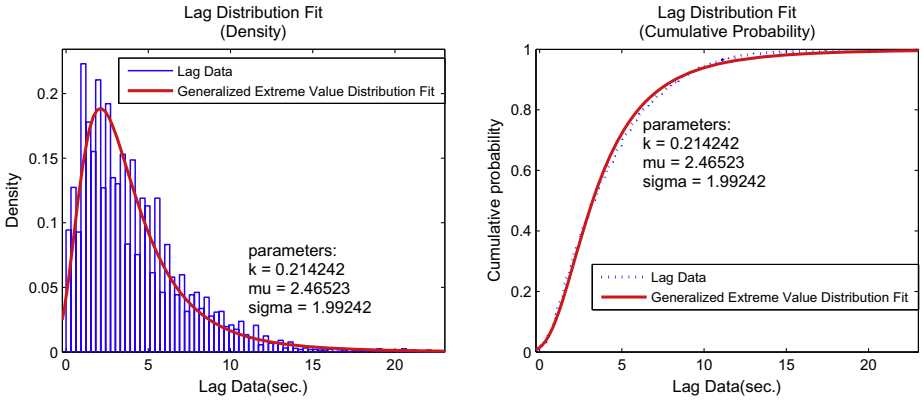
(e) Subgroup1 Size Over Group Size(Size 32)

(f) Subgroup1 Size Over Group Size(Size 48)

Fig. 4. Characteristics of live streaming data requests.

where $\lambda = 1 + \kappa\frac{x-\mu}{\sigma}$, $\lambda > 0$ and κ, σ, μ are the shape, scale, and location parameters, respectively.

Fig. 5 shows that the lag length is about 15 s, which cover more than 95% of the overall data requests. The lag distribution describes the lifetime of a data piece. When a data piece is first released, it is requested by only a few peers. Then it becomes very hot because more and more data copies are available in the P2P network and the playback position of most users moves to this data piece. However, as the playback window forwards, the data piece becomes unpopular again, and is obsolete after about 15 s. The findings are inspiring in caching algorithm design. For a channel with average group size of $32 * 0.25 + 48 * 0.75 = 44$ (refer to Section 4.1) data pieces, the playback rate is about $44 * 1$ KBytes $* 8$ bits/Byte $* 1$ s = 352 kbps since the typical data piece size is



(a) Probability Density View (b) Cumulative Probability View

Fig. 5. Lag distribution fit.

1 KBytes. So the cache needed for a channel with 352 kbps playback rate is about 44 * 1 KBytes * 15 = 660 KBytes. If cached data is carefully updated and all the current active data pieces are cached, a small cache can achieve nearly 100% hit rate.

So far, we have investigated the request characteristics of P2P live streaming, assuming all the requested data is fetched from the Internet rather than the local regional network. Because of the poor locality discovery mechanism, the assumption is reasonable and also proved by our inspection on real traces. But there's still possibility that data is fetched from the local regional network especially if the streaming content is static and data source locates in the same regional network. We argue that different data pieces have the same probability to be fetched locally, so the request characteristics observed in the gateway won't change dramatically, although much less data requests can be seen.

4.3. Channel popularity

Similar to the role of object popularity in P2P file sharing caching, *channel popularity* (i.e. the online user number of a channel) is important for live streaming in caching algorithm design. Channel popularity varies when user joins the overlay network or aborts connection. When designing caching algorithms for P2P live streaming, cache should be allocated according to channel popularity.

To study channel popularity, we measure the online user number of totally 4403 channels at 21:30 pm on 2008-11-07 with the help of PPLive. The online user number versus channel rank distribution is plotted in the log–log scale in Fig. 6. The distribution can be modeled as a line with slope of about 0.53. The relationship is expressed as

$$\log y = \alpha \log x + \beta, \tag{2}$$

where $0 \leqslant \log x \leqslant \gamma$, $\alpha = -0.53$, $\beta = 3.73$, $\gamma = 3.0$, y is the online user number and x is the channel rank.

We have also modeled the distribution with stretched exponential distribution [19], but found that although the distribution fits the unpopular channels well, it is not so good for the top 20 channels. Because popular channels account for larger fraction of network traffic, we prefer to model the top 100 channels with a more accurate linear relationship.

Eq. (2) can be used as a guideline in deciding channel user number for generating synthetic data requests. Given the channel number, we can pick points evenly on the line so that both popular and unpopular channels are generated proportionally.

4.4. Data request synthesis

We have studied request characteristics for single users, the lag distribution among peers in a channel and channel popularity. Now we show how to utilize the findings to generate synthetic data requests for caching algorithm evaluation. There are two objectives when generating synthetic requests. One is to match real traces as well as possible. The other is to follow the typical patterns. We achieve these goals by strictly following the models in previous sections and adding randomness if necessary. The synthetic data request generator algorithm is described in Fig. 7.

The procedure involves six steps:

1. *Determine online user number for each channel (Fig. 7, Line 1,2).* Given the channel number *chN*, first we determine the online user number for each channel. Accord-

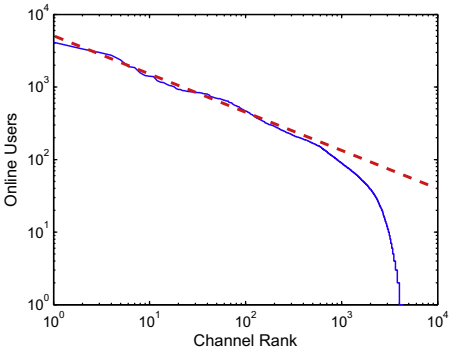


Fig. 6. Channel popularity modeling.

Data Requests Generator

input:
chN: channel number
groupSize: group size
duration: playback duration (seconds)
output:
requests for *chN* channels, lasting *duration* seconds
Requests Generator:
1. **for** each channel *ch* from 1 to *chN*
2. online user $chArr[ch] = 10^{\alpha*(\gamma/chN)*ch+\beta}$ // see Equation 2
3. **for** each channel *ch* from 1 to *chN*
4. **for** each user *user* from 1 to *chArr*[*ch*]
5. generate lag array *lagArr*[*ch*][*user*] using matlab // see Figure 5, Equation 1
6. **for** each channel *ch* from 1 to *chN*
7. **for** each user *user* from 1 to *chArr*[*ch*]
8. generate requests for *user* lasting *duration* seconds // see Figure 4, Section 4
9. merge and sort all the requests using time stamp as the key

Fig. 7. Data request generator.

ing to the channel popularity relationship in Eq. (2), we generate a series of online user numbers (y) by picking rank values ($\log x$) evenly on the x-axis.

2. *Generate lag data (Fig. 7, Line 3,4,5).* As discussed in Section 4.2, peers of the same channel have a certain lag between each other. The request lags obey the GEV distribution (Fig. 5), so we use matlab to generate request lags for each channel and assign a lag to each user as the initial request time stamp. The distribution parameters are marked in Fig. 5.So far, the online user number in each channel and the initial request time stamp for each user have been determined. The following two steps are for generating requests for each user in step 5.
3. *Determine request interval.* As shown in Section 4.1, request intervals are classified into two types: request interval between groups/subgroups and request interval inside subgroups. We use 1 s as request interval between groups and 0.5 s between subgroups (Fig. 4a). To match real traces better, some randomness is added(e.g., add $-0.05 \sim +0.05$ for the group/subgroup interval). Request interval inside subgroups is 0–0.0005 as shown in Fig. 4d. Note that the request interval is calculated when generating every new request.
4. *Determine subgroup size.* Group size reflects the requesting rate. Higher requesting rate means more requests are sent in each group, thus larger group size. In our case, groups of size 32 and 48 dominate (Fig. 4b and c). So only these two sizes are used. The probability for group size of 32 is 0.25 and 0.75 for 48 (Fig. 4c). Whether requests of the same group are sent in subgroup1 or subgroup 2 is determined according to Fig. 4e and f in a probabilistic fashion.
5. *Generate request data for each user (Fig. 7, Line 6,7,8).* We assign a lag generated in step 2 as the initial time tamp to each user and generate requests ([time stamp, channel ID, sequence number] entry) using steps 3,4. The sequence number starts from 1, monotonically increasing. Steps 3,4 are executed iteratively until time stamp exceeds *duration* seconds.
6. *Merge all the requests and get output (Fig. 7, Line 9).* Every request has its time stamp, which indicates its sending time. We merge all the requests by sorting them with the key of time stamp. The output data contains the (time stamp, channel ID, sequence number) entry for each request, which is the final synthetic data.

The synthetic data generation process is almost online except that the lag between requests must be generated with matlab. Since it's very convenient to generate data given a distribution with matlab, we decide to do it off-line after balancing the efforts and gains of implementing a new distribution data generating algorithm. Also generating data with matlab makes the process easier-to-understand. But we can also do it online, which is actually an implementation issue.

5. Sliding window (SLW) caching algorithm

Based on the observations in previous sections, we propose a novel caching algorithm for P2P live streaming called sliding window (SLW) algorithm. SLW caches the hottest data pieces in the requesting window and distributes cache among channels according to their popularity. Although P2P live streaming traffic is very favorable for caching (see Section 4.2), it's unnecessary and wasteful to cache all the active data pieces of all channels because most of the channels are not popular. Our caching objective is to reduce the traffic on the link. So a well-designed caching algorithm should cache data pieces of popular channels preferentially.

The caching unit of SLW is data piece since it's the data transmission unit in P2P live streaming. Different data pieces are usually fetched from different peers. The data piece size is fixed (1KB) in PPLive [29].

The basic idea of SLW is to maintain a *caching window* storing the most popular data pieces in the requesting window for each channel. At any given time, all the data pieces in the requesting window are active, but some are requested more frequently. As data pieces are released gradually and data piece sequence number increases monotonically, these popular data pieces are continuous in the requesting window according to the lag distribution in Section 4.2. SLW stores these data pieces in cache which are called the *caching window* of a channel. The caching window size should not be larger than the requesting window size, or it is a waste of cache. We denote the requesting window size as Ω, and it can be calculated by

$$\Omega = R * L. \tag{3}$$

where R is the requesting rate and L is the lag length. Data piece is the unit of the cache size, the caching window size and the requesting window size.

The requesting window is sliding forward, so the caching window should also slide forward to accommodate the popularity change of data pieces. That's why we call our

new algorithm the *sliding window* algorithm. To decide whether the caching window should be moved forward, we sample the request count of the head and tail parts of the caching window. If the data pieces at the head are more popular than those at the tail, we know it's time to admit new data pieces into cache and evict those at the tail. In this way, it's guaranteed that the most popular data pieces are always cached. The pseudo-code of the SLW algorithm is shown in Fig. 8.

The algorithm "Cache Replacement" in Fig. 8 describes the process of sliding the caching window, while the algorithm "Channel Cache Size Adjustment" updates the cache allocation among channels periodically according to the trend of channel popularity. The channel popularity statistics are maintained in line 2 and reset after each adjustment. They reflect not only the number of online users but also the channel requesting rate. Cache is allocated proportionally among channels according to their popularity statistics. The upper limit of the allocated cache for each channel is its requesting window size Ω. We distribute cache among the channels according to the data request number instead of online user number of each channel because the data request number is more accurate with regard to the different requesting window sizes for different channels.

Compared with the typical LRU (Least Recently Used) algorithm, which utilizes temporal popularity by caching the most popular data pieces, SLW also exploits spatial locality by maintaining a continuous caching window. Although the popularity of the data pieces in the caching window may vary temporarily, which can be reflected by LRU immediately, the caching window does not move forward until the data pieces at the head are more popular than those at the tail. So the employment of spatial locality helps resist temporary popularity fluctuations.

The SLW algorithm has several advantages.

SLW Caching Algorithm

Cache Replacement:

1. **for** each request $< ch, piece >$
2. $chHot[ch] + +, totalChHot + +$
3. **if** $piece >= lower[ch] \&\& piece <= upper[ch]$
4. cache hit
5. **if** *piece* belongs to the tail part of caching window
6. $tailHot + +$
7. **else if** *piece* belongs to the head part of caching window
8. $headHot + +$
9. **if** $headHot > tailHot$
10. reset *headHot* and *tailHot*
11. $lower[ch] += offset, upper[ch] += offset$
// forward the window by $offset$

Channel Cache Size Adjustment:

12. sort channel hot in reverse order
13. **for** each channel *ch*
14. $cacheSize[ch] = totalSize * chHot[ch]/totalChHot$
15. **if** $cacheSize[ch] > \Omega$
16. $cacheSize[ch] = \Omega$
17. $totalChHot -= chHot[ch]$
18. $totalSize -= cacheSize[ch]$
// cache size Ω is enough

Fig. 8. The SLW caching algorithm.

- Flexible. Section 4.2 shows that the request lag follows the GEV distribution, but the parameters may vary in different network environments or time periods. SLW employs the distribution indirectly by sampling the data popularity in the head and tail parts, irrelevant to the specific parameters of the GEV distribution. Besides, it's common in a P2P network that peers join the network or abort connections frequently. SLW accommodates channel popularity variation by maintaining channel popularity statistics and adjusting cache size accordingly.
- Skillfully using the characteristics of P2P live streaming traffic. SLW integrates information regarding all aspects including channel popularity, temporal and spatial locality. SLW always caches the most popular data pieces in the playback window and slides ahead when data pieces at the head become more popular. These cached data pieces are always continuous and accesses to the most popular data pieces indicate the increase of accesses to the data pieces of the head part. Moreover, as new data pieces get popular very quickly (Fig. 5), they are cached very soon once appear, which speeds up data delivery.
- Computationally-efficient. As Fig. 8 shows, the overhead of cache management is low. The time complexity for cache replacement is $O(1)$, and the space complexity is $O(chN)$. The periodic adjustment of channel cache size has $O(chN)$ both as the time and space complexity.

6. Evaluation and discussion

In this section, we use synthetic data requests to evaluate the performance of our SLW algorithm, and compare it with three typical web caching algorithms (LRU, LFU, GDS) and two caching algorithms (LSB, P2P) proposed for P2P file sharing. The usual FIFO and OPT algorithms are also implemented for comparison. Then we show how to calculate the theoretical performance curve of the SLW algorithm and standardize it as guidelines for estimating SLW performance.

6.1. Experimental setup

The main objective of caching P2P live streaming traffic is to reduce extra traffic on the network link. So the most important metric of the caching algorithms is the cache hit rate, i.e. the percentage of data requests served locally. Usually the cache size is based on the unit of bytes. As the cache eviction and admission are based on data piece for live streaming, we employ the data piece number to denote cache size.

We use synthetic data requests generated in Section 4.4 to evaluate caching algorithms. Some typical parameters of the data set are listed in Table 2. Note that all the ten channels have the same lag length, group size and duration values.

We compare the performance of SLW with that of five other algorithms. Implementation details of some caching algorithms are modified to accommodate the P2P live

Table 2
Synthetic data requests parameters.

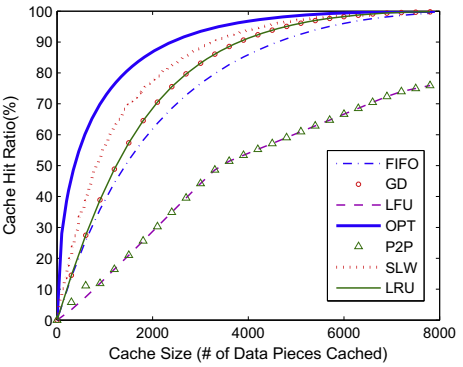
Channel number (*chN*)	10
Lag length (*L*)	15 s
Group size (*groupSize*)	32 for probability 0.25 48 for probability 0.75 44 data pieces per second on average
Duration (*duration*)	3000 s

streaming traffic. The algorithms are implemented as follows:

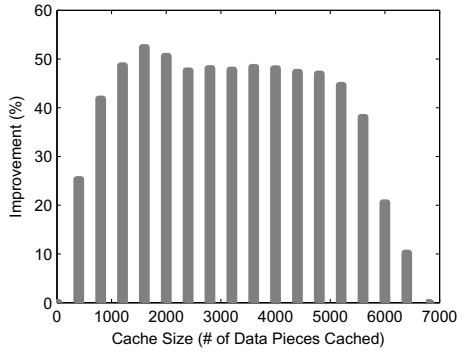
- *OPT*: The OPT (Optimal) algorithm eliminates the data pieces that will be latest used. It serves as a benchmark for the performance of other algorithms. The result of OPT is the upper bound which can hardly be achieved by other algorithms since OPT uses future information which is not known a priori in practice.
- *FIFO*: The FIFO (First In First Out) algorithm evicts data pieces in the first-in-first-out fashion. It's the first algorithm that comes to mind when we find the sliding window characteristics of the traffic.
- *LRU*: The LRU (Least Recently Used) algorithm evicts the least recently used data pieces. It explores the temporal locality of data and is widely used.
- *LFU-LSB*: The LSB (Least Sent Bytes) algorithm uses the transmitted bytes of an object as eviction criteria [10]. The LFU (Least Frequently Used) algorithm evicts the object with the least access frequency. As the request frequency and served bytes correlate with each other in live streaming, these two algorithms are implemented as the same: first look for a channel least frequently accessed, and then eliminate the data piece least requested.
- *GD*: As the GDS (GreedyDual-Size) algorithm [30] takes object size into consideration, which cannot be directly applied for live streaming, we implement the more basic GD algorithm [31]. The GD algorithm assigns a weight to each newly cached data piece. When cache miss occurs, the data piece with the least weight is removed, and the weights of the other pieces in cache are reduced by the least weight. If a data piece is hit, its weight restores to the original value. The mechanism of the GD algorithm is very similar to that of LRU.
- *P2P*: The P2P algorithm [6] is proposed for P2P file sharing. It evicts the object with the lowest γ_i, where γ_i is updated by "$\gamma_i+=$ hit/cached size of object *i*". For P2P live streaming, the algorithm is implemented to eliminate the least frequently requested data piece from the channel with the lowest γ_i, and γ_i is updated with "$\gamma_i+=$ hit/cached size of channel *i*". So cache eviction always occurs in the channel with least requests, like LFU.

6.2. Performance results

The examined algorithms fall into five categories as previously illustrated: OPT, SLW, LRU, FIFO and LFU. We



(a) Performance Results



(b) Performance Gain of SLW over LRU

Fig. 9. Algorithms comparison.

can also see the grouping in their performance results (Fig. 9). LFU is not suitable for live streaming because data piece popularity is changing all the time. The frequent access of some data pieces especially those at the tail of the requesting window does not indicate that they will be requested again. Temporal locality is observed from the good performance of LRU. But it does not utilize the continuity of popular data pieces (i.e., spatial locality), so its performance is not so good as SLW. SLW works best among the online algorithms since it explores both temporal and spatial locality. The performance of SLW is the closest to that of the off-line OPT algorithm, gaining nearly 50% improvement over the performance gap between LRU and OPT (Fig. 9). The poor performance of FIFO implies that simply applying the FIFO policy cannot achieve very good results.

Fig. 9 verifies that P2P live streaming traffic is very favorable for caching. In our case, the requesting window size (Ω) of a single channel is 660. When cache size is about 6600 ($chN * \Omega$), the cache hit rate of SLW nearly reaches 100%. So it is with the OPT, GD, LRU and FIFO algorithms. The sharp increase of cache hit rate with the increase of cache size is considerable when cache size is small. Only about 2000 data pieces need to be cached to serve 80% of the data requests. That's to say, deploying a small cache in the network can significantly reduce the traffic imposed by P2P live streaming systems.

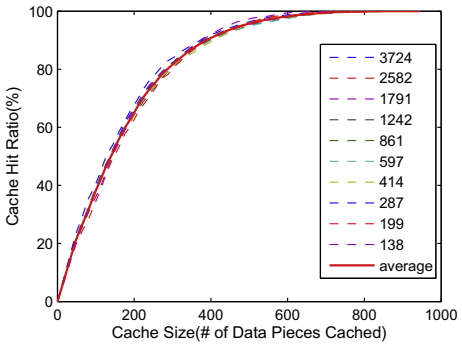


Fig. 10. Online user number affecting SLW performance.

To examine the effect of online users on the SLW performance, we run SLW through the synthetic data in each channel with different online user numbers ranging from 138 to 3724. The experimental results (Fig. 10) show that the SLW performance on a single channel is irrelevant to the online user number and the cache hit rate is nearly 100% when cache size is as large as $\Omega = 44 * 15 = 660$. It's because the cache size determines the fraction of requests that can be cached in the requesting window. If the requesting window size is the same, the cache hit rate depends on the cache size only.

It's common that different channels may have different requesting rates. We run SLW on a data set with average requesting rate of 76 data pieces per second to study the requesting rate impact on its performance. The new data set has the requesting window size $\Omega = 76 * 15 = 1140$. Fig. 11 shows the SLW performance on this data set. We normalize the performance curve linearly to that of the data set with group size 44 and find that the two curves match very well (Fig. 11). The experimental results verify that the requesting rate correlates linearly with the performance of SLW. It implies that the same cache performance can be maintained by linearly increasing requesting window size while requesting rate increases.

6.3. Modeling the SLW algorithm performance

In this section, we try to answer the question: *how to estimate the SLW algorithm performance on a specific data set given a cache size?* We calculate the cache hit rate theoretically with the GEV distribution and then compare it with the experimental SLW performance. Results show that the theoretical performance and the experimental one matches perfectly. Consequently the performance profiles by theoretical computation can be employed to estimate the SLW performance, which is shown at the end of this section.

We have obtained the experimental performance of SLW. Now we try to calculate its performance theoretically. The point-in-time data requests of one channel are studied. Ideally the cache hit rate for any given time is the same since the data requesting rate is stable. So the cache hit rate calculated with a snapshot can represent the overall algorithm performance. The GEV distribution describes the data piece popularity during its lifetime. We assume the first data requests for every two adjacent data pieces are released with the same interval length, which is reasonable with stable requesting rate. Consequently all the data pieces in the requesting window correspond to the points (i.e. the age of each data piece) evenly distributed in the x-axis span (Fig. 12) and their current popularity can be reflected by the corresponding y values of the points on the distribution curve. Popular data pieces have more requests while unpopular ones have less. In other words, the discrete x value can be viewed as a data piece in the requesting window, and the corresponding y value reflects the proportional population requesting the data piece. As SLW always caches the most popular data pieces, we can calculate the shadowed area in Fig. 12 as the cache hit rate and formulate it (h) as

$$h = F(x_2) - F(x_1), \tag{4}$$

with the constraints

$$f(x_2) = f(x_1), \quad x_1 + t = x_2, \quad x_1 \leqslant \mu_0 \leqslant x_2,$$

where $F(x)$ and $f(x)$ are the cumulative distribution function and probability density function respectively. The function value $f(x)$ reaches its maximum when $x = \mu_0$. Variable t is the cache size. We need to obtain the relationship between h and t.

For each x_1, we can calculate x_2 (thus t) with binary search in x ($x > \mu_0$) on the GEV distribution curve in

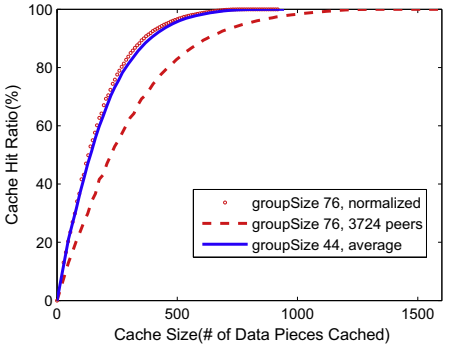


Fig. 11. Group size affecting SLW performance.

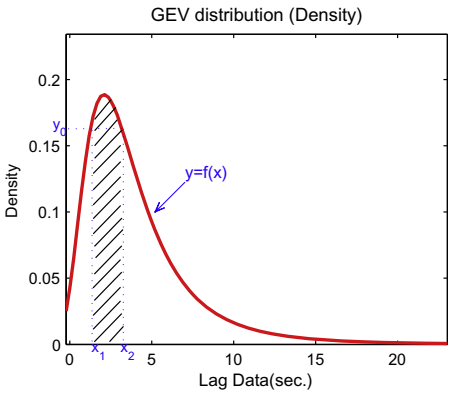


Fig. 12. Cache hit ratio calculation.

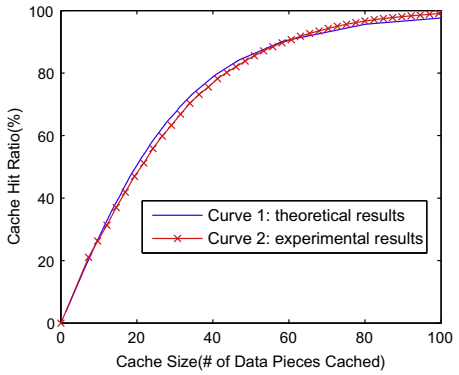


Fig. 13. Theoretical vs. experimental results.

Fig. 12. Then h can be calculated with Eq. (4). By enumerating x_1, we can obtain a list of (h,t) pairs. We multiply $100/L$ to every t to normalize the results from requesting window size $\Omega = L$ ($L = 15$ here) to $\Omega = 100$ (100 is chosen just for convenience), and plot the new (h,t) pairs in Fig. 13(Curve 1), which is the theoretical SLW performance. The x-axis indicates the percentage of data pieces cached over the requesting window, and the y-axis indicates the percentage of data requests served from cache. To compare the theoretical performance with the experimental one, we also normalize the experimental performance curve to $\Omega = 100$ (Curve 2 in Fig. 13) and find that these two curves almost coincide with each other.

The theoretical performance curve, only relevant to the requesting window property decided by the GEV distribution parameters, matches the experimental results very well. We call it a standard profile. By theoretically calculating the cache hit rate for some specific distributions, we obtain some standard profiles shown in Fig. 14. Table 3 lists the distribution parameters. The different performance curves reflect the different properties of different distributions.

Now we can use these standard profiles as guidelines to estimate the SLW performance given a cache size. First distribute cache size among the channels according to their channel popularity and data requesting rates (refer to the "Channel Cache Size Adjustment" routine in Fig. 8). And then use the requesting rate and the standard profiles to compute cache hit rate of each channel. At last, the overall cache hit rate can be calculated by adding the cache hit rate of single channels weighted by its proportional popularity.

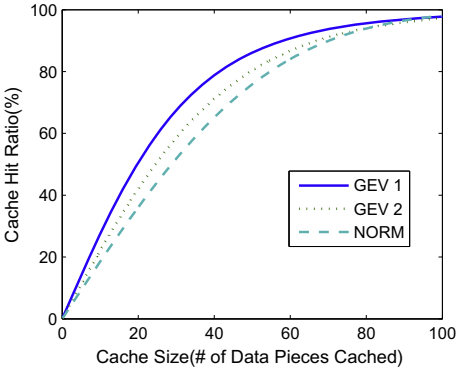


Fig. 14. Comparison of different distributions.

Table 3
Different distribution parameters.

	Distribution	Parameters	Lag length
GEV 1	GEV	$\kappa = 0.214242$ $\mu = 2.46523$ $\sigma = 1.99242$	15 s
GEV 2	GEV	$\kappa = 0.1, \mu = 10, \sigma = 14$	85 s
NORM	normal	$\mu = 7.5, \sigma = 3.2$	15 s

6.4. Cache size optimization

We can also estimate the required cache size to achieve certain cache hit rate. Since the cache hit rate of SLW increases with the increase of cache size, we can use binary search to estimate the required cache size with a given cache hit rate. The estimation can help avoid unnecessary cache usage and optimize cache size.

For example, the cache hit rate of SLW nearly reaches 100% when cache size is about 6600 in Fig. 9. Actually we can calculate the required cache size with any given cache hit rate. The calculated cache size is also the minimum one to reach certain cache performance. Extra cache can be spared if the performance already meets real requirements.

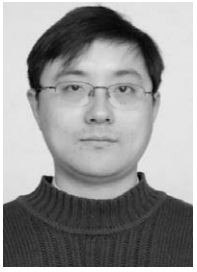
7. Conclusion

In this paper, we studied the characteristics of data requests in P2P live streaming and modeled the lag distribution with the generalized extreme value distribution. With the help of the findings, we designed a data request generator to generate synthetic traffic for P2P live streaming applications. Furthermore, we proposed a novel caching algorithm for P2P live streaming applications-SLW. The SLW algorithm explores both temporal and spatial locality of data requests and gets the best performance among the online caching policies including LRU, LFU and FIFO. Experiments prove that the P2P live streaming traffic is very favorable for caching. Deploying a small cache in the network can significantly reduce the traffic imposed by the P2P live streaming systems. By studying the factors affecting SLW performance, we calculated standard performance profiles with the lag distributions and illustrated their practical usage in estimating algorithm performance. In the future, we plan to make a similar study on the caching strategies for P2P video-on-demand traffic. Since P2P VoD traffic does not have obvious synchronization characteristics as P2P live streaming, caching strategies for P2P VoD traffic may share more similarities with those for P2P file

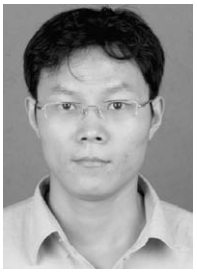
sharing traffic. The effectiveness and benefits of the SLW algorithm in real environments need further investigation as well.

References

[1] <http://www.bittorrent.com>.
[2] <http://www.emule.org>.
[3] <http://www.skype.com>.
[4] <http://www.pplive.com>.
[5] T. Karagiannis, A. Broido, N. Brownlee, K.C. Claffy, M. Faloutsos, Is P2P dying or just hiding? in: Proceedings of the GLOBECOM 2004 Conference, IEEE Computer Society Press, Dallas, Texas, 2004.
[6] O. Saleh, M. Hefeeda, Modeling and caching of peer-to-peer traffic, in: ICNP '06: Proceedings of the 2006 IEEE International Conference on Network Protocols, IEEE Computer Society, Washington, DC, USA, 2006, pp. 249–258. doi:http://dx.doi.org/10.1109/ICNP.2006.320218.
[7] L. Breslau, P. Cao, L. Fan, G. Phillips, S. Shenker, Web caching and zipf-like distributions: evidence and implications 1 (1999) 126–134, doi:10.1109/INFCOM.1999.749260.
[8] S. Podlipnig, L. Böszörmenyi, A survey of web cache replacement strategies, ACM Comput. Surv. 35 (4) (2003) 374–398. doi:<http://doi.acm.org/10.1145/954339.954341>.
[9] J. Liu, J. Xu, Proxy caching for media streaming over the internet communications magazine, IEEE 42 (8) (2004) 88–94, doi:10.1109/MCOM.2004.1321397.
[10] A. Wierzbicki, N. Leibowitz, M. Ripeanu, R. Wozniak, Cache replacement policies revisited: the case of P2P traffic, in: Proceedings of the 2004 IEEE International Symposium on Cluster Computing and the Grid, 2004, pp. 182–189. doi:10.1109/CCGrid.2004.1336565.
[11] J. Wang, A survey of web caching schemes for the internet, SIGCOMM Comput. Commun. Rev. 29 (5) (1999) 36–46. doi:<http://doi.acm.org/10.1145/505696.505701>.
[12] R.J. Dunn, Effectiveness of caching on a peer-to-peer workload, Master's Thesis, University of Washington, Seattle, 2002.
[13] Y. Chen, C. Chen, C. Li, A measurement study of cache rejection in p2p live streaming system, in: ICDCSW '08: Proceedings of the 2008 the 28th International Conference on Distributed Computing Systems Workshops, IEEE Computer Society, Washington, DC, USA, 2008, pp. 12–17.
[14] S. Deshpande, J. Noh, P2P live streaming of video in peer-to-peer systems, in: ICME, IEEE, 2008, pp. 649–652. <http://dblp.uni-trier.de/db/conf/icmcs/icme2008.html#DeshpandeN08>.
[15] X. Hei, C. Liang, J. Liang, Y. Liu, K. Ross, A measurement study of a large-scale P2P iptv system, IEEE Transactions on Multimedia 9 (8) (2007) 1672–1687, doi:10.1109/TMM.2007.907451.
[16] S. Ali, A. Mathur, H. Zhang, Measurement of commercial peer-to-peer live video streaming. <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1>.
[17] N. Leibowitz, A. Bergman, R. Ben-shaul, A. Shavit, Are file swapping networks cacheable? characterizing P2P traffic, in: Proceedings of the 7th International WWW Caching Workshop, 2002.
[18] T. Karagiannis, P. Rodriguez, K. Papagiannaki, Should internet service providers fear peer-assisted content distribution? in: IMC '05: Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement, USENIX Association, Berkeley, CA, USA, 2005, pp. 6–6.
[19] L. Guo, E. Tan, S. Chen, Z. Xiao, X. Zhang, Does internet media traffic really follow zipf-like distribution? in: SIGMETRICS '07: Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, ACM, New York, NY, USA, 2007, pp. 359–360. doi:<http://doi.acm.org/10.1145/1254882.1254929>.
[20] S. Chen, H. Wang, X. Zhang, B. Shen, S. Wee, Segment-based proxy caching for internet streaming media delivery, IEEE MultiMedia 12 (3) (2005) 59–67. doi:http://dx.doi.org/10.1109/MMUL.2005.56.
[21] S. Chen, B. Shen, S. Wee, X. Zhang, Designs of high quality streaming proxy systems, in: Proceedings of IEEE, 23th Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM, vol. 3, 2004, pp. 1512–1521. doi:10.1109/INFCOM.2004.1354565.
[22] S. Sen, J. Rexford, D. Towsley, Proxy prefix caching for multimedia streams, in: Proceedings of IEEE, 18th Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM, vol. 3, 1999, pp. 1310–1319. doi:10.1109/INFCOM.1999.752149.
[23] K.-L. Wu, P.S. Yu, J.L. Wolf, Segment-based proxy caching of multimedia streams, in: WWW '01: Proceedings of the 10th International Conference on World Wide Web, ACM, New York, NY, USA, 2001, pp. 36–44. doi:<http://doi.acm.org/10.1145/371920.371933>.
[24] M. Hefeeda, C. Hsu, K. Mokhtarian, Design and evaluation of a proxy cache for peer-to-peer traffic, Tech. Rep., Simon Fraser University (Sep 2008).
[25] <http://www.cachelogic.com>.
[26] X. Zhang, J. Liu, B. Li, Y.-S. Yum, Coolstreaming/donet: a data-driven overlay network for peer-to-peer live media streaming, in: Proceedings of IEEE, 24th Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM, vol. 3, 2005, pp. 2102–2111. doi:doi:10.1109/INFCOM.2005.1498486.
[27] <http://www.ppstream.com>/.
[28] Extreme value distribution. URL: <http://www.mathwave.com/articles/extreme-value-distributions.html>.
[29] Y. Huang, T.Z. Fu, D.-M. Chiu, J.C. Lui, C. Huang, Challenges, design and analysis of a large-scale P2P-vod system, in: SIGCOMM '08: Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication, ACM, New York, NY, USA, 2008, pp. 375–388. doi:<http://doi.acm.org/10.1145/1402958.1403001>.
[30] P. Cao, S. Irani, Cost-aware www proxy caching algorithms, in: USITS'97: Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems, USENIX Association, Berkeley, CA, USA, 1997, pp. 18–18.
[31] N. Young, The k-server dual and loose competitiveness for paging, Algorithmica 11 (1994) 525–541.

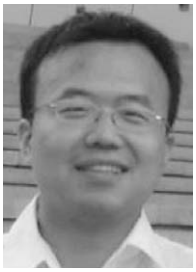


Ke Xu was born in Jiangsu, PR China, in 1974. He received the B.S., M.S. and Ph.D. degrees in computer science from Tsinghua University, China in 1996, 1998 and 2001, respectively. Currently, he is an Associate Professor in the department of computer science of Tsinghua University. His research interests include next generation Internet, switch and router architecture, P2P and overlay network. He is a member of ACM SIGCOMM, member of IEEE and IEEE Communication Society.



Ming Zhang is currently a M.S. candidate of School of Software, Tsinghua University. His research interests include network traffic classification and applications, peer-to-peer systems and networking security.

Jiangchuan Liu received the BEng degree (cum laude) from Tsinghua University, Beijing, China, in 1999, and the Ph.D. degree from The Hong Kong University of Science and Technology in 2003, both in computer science. He is currently an Associate Professor in the School of Computing Science, Simon Fraser University, British Columbia, Canada, and was an Assistant Professor in the Department of Computer Science and Engineering at The Chinese University of Hong Kong from 2003 to 2004.
His research interests include multimedia systems and networks, wireless ad hoc and sensor networks, and peer-to-peer and overlay networks. He is an Associate Editor of IEEE Transactions on Multimedia, and an editor of IEEE Communications Surveys and Tutorials. He is a Senior Member of IEEE and a member of Sigma Xi.



Zhijing Qin is currently a Master student of School of Software and Microelectronics, Peking University. His research interests include traffic identification, P2P live streaming and network pricing.

Mingjiang Ye received the B.Eng. degree in computer science from Tsinghua University, China, in 2000. He is currently a Ph.D. candidate in Department of Computer Science and Technology, Tsinghua University, China. His research interests are in the filed of Internet technologies and applications, including network traffic management, network traffic classification and peer-to-peer systems and overlay networks.