Towards Low-Redundancy Push-Pull P2P Live Streaming

Zhenjiang Li, Yao Yu, Xiaojun Hei and Danny H.K. Tsang

Department of Electronic and Computer Engineering
The Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong
lzjiang@ust.hk, eeallan@ust.hk, heixj@ece.ust.hk, eetsang@ece.ust.hk

Abstract

P2P live streaming systems are developed in two major approaches: tree-push versus mesh-pull. The hybrid push-pull streaming, as an emerging and promising approach, offers a good tradeoff between traffic overhead and system throughput. In this paper, we demonstrate that video redundancy is a large contributor of the traffic overhead in push-pull systems. To reduce the traffic overhead, we propose simple but effective sub-stream scheduling and re-scheduling mechanisms, implemented in a push-pull streaming prototype called Low-redundancy Streaming (LStreaming). To demonstrate its effectiveness on reducing the traffic overhead, we conduct both simulation and prototype experiments and compare the proposed LStreaming with random mesh-pull and GridMedia. The simulation results show that LStreaming significantly reduces the total traffic overhead, i.e., up to 33% and 37% reduction compared with mesh-pull and GridMedia in dynamic P2P environments. LStreaming also achieves the throughput, more close to the optimal than the other two schemes, and sustains a better video playback quality. The prototype experiments show that LStreaming is practical and achieves the expected performance.

Index Terms

peer-to-peer live streaming, push-pull, tree-push, mesh-pull

I. Introduction

The emerging peer-to-peer (P2P) networks have appeared to be the most promising driving force for video streaming over the Internet [1]–[3]. To date, P2P video streaming systems have enjoyed a number of large-scale deployments, notably, CoolStreaming [2], PPLive [4] and many others. P2P streaming architectures have advanced significantly in two major approaches: tree-push versus mesh-pull. The mesh-pull systems apply a simple design principle and achieve inherent robustness particularly desirable adaptability for highly dynamic, high-churn P2P environment [5]. However, these mesh-pull systems often suffer from high traffic overhead, long start-up delays, significant video channel switching delays and large peer playback time lags [6]. Unlike mesh-pull systems, tree-push systems may achieve high throughput, low overhead and small delay if the tree structure does not break down

due to peer churn and peers at the higher level of the tree have sufficient upload capacities to support streaming for their children peers.

Recently, researchers are exploring a new class of hybrid push-pull architecture, promising to offer a good tradeoff among system throughput, scheduling overhead and the delay performance [7], [8]. In this paper, we study this push-pull streaming architecture in detail. We find that this emerging push-pull streaming approach may incur significant video redundancy if the system is not carefully designed.

An ideal live streaming system should be practical and robust, in addition to high throughput and low overhead. Mesh-pull and tree-push are complementary in some aspects and a joint design may lead to meet these system requirements. A small number of push-pull streaming systems have achieved some success. GridMedia [7] and CoolStreaming+ [8] are enhanced with the push-pull streaming feature. In [7], the simulation results have showed the push-pull approach has the potential to achieve the optimal throughput and reduce the signaling overhead compared with the mesh-pull scheme. We examine the traffic overhead of GridMedia and find that the traffic overhead may reach as high as 10% of the effective video traffic. Due to the large video traffic volume, we believe that 10% overhead is quite large and this overhead may be reduced significantly. The total traffic overhead consists of signaling traffic and redundant video traffic. Among the total traffic overhead, we find that the redundant video traffic contribute the major portion in push-pull systems.

A generic push-pull streaming system works as follows. Each peer starts with the mesh-pull mode. After some initial time, it requests certain neighbors to push chunks of sub-streams; then, the tree-push mode is enabled. To achieve high throughput, low overhead and robustness, the mesh-pull mode is still working to serve as a backup to download those missing chunks when their playback deadline is approaching. To combat network dynamics and peer churn, each peer should select the most "powerful" peers to push sub-streams and switch from one "bad" neighbor to a "good" neighbor. Different push-pull systems may vary significantly in peer switching. We call this behavior "push-neighbor switching". We find that this push-neighbor switching is the dominating factor of redundant video traffic. The essential reason behind the push-neighbor switching is the signaling asynchronization between the peer and its push-neighbors, due to the propagation delay and the chunk buffering delay. This asynchronization cannot be completely avoided and is not a problem by itself if there is no push-neighbor switching. Therefore, the key issue is to reduce the frequency of the push-neighbor switching in push-pull systems.

In order to reduce the total overhead especially the video redundancy, and to achieve an optimal throughput, we propose two simple but effective mechanisms, implemented in a prototype system called Low-redundancy Streaming (LStreaming). Our contribution in this paper can be summarized as follows.

- We demonstrate that video redundancy is an important issue in push-pull P2P live streaming. The solution is

to reduce the frequency of push-neighbor switching.

- We propose simple but effective learning-based sub-stream scheduling and re-scheduling mechanisms to reduce the video redundancy in LStreaming.

- We evaluate the performance of LStreaming using simulation and prototype. The simulation results show that LStreaming reduces the total overhead significantly by reducing the video redundancy. At the same time, LStreaming maintains an optimal throughput in both static and dynamic P2P environments. The prototype experiments show that LStreaming is practical and achieves the expected performance.

The rest of this paper is organized as follows. In Section II, we outline the system components of LStreaming. We present its design and implementation in Section III and report the performance evaluation using simulation and prototype in Section IV. Finally, the concluding remarks are made in Section V.

II. ARCHITECTURE OVERVIEW

A. Peer management

In LStreaming, a video is divided into data chunks. Each chunk is indexed with a unique increasing identifier and is streamed out from a source server for distribution. Chunks, which are cached in a peer's buffer, are depicted using buffer maps. A buffer map includes the smallest ID of the cached chunks, the width of the buffer map, and a string of zeroes and ones, which indicate the cached chunks available for sharing. All the peers, who are interested in this video, form a channel group and help each other to deliver the video. The peers of one channel are managed by a tracker server. When a new peer joins a channel, it first contacts the tracker server to report its arrival, obtains a peer list of this channel and randomly selects some peers as its neighbors. Between two peers, buffer maps are exchanged periodically in LStreaming. Based on the buffer maps, a peer downloads missing video chunks from its neighbors.

B. Sub-stream creation

In mesh-pull streaming, video is downloaded in a chunk-by-chunk fashion. The basic streaming process follows a repeating pattern of exchanging buffer maps and then downloading missing chunks [7]. Significant signaling overhead may incur during this process. To download one chunk, potentially at least one pull-based request is issued. One possible method to reduce the signaling overhead is to request one sub-stream instead of one chunk on each request. As shown in Fig. 1, the original video stream is segmented into S sub-streams. As long as the connection between two peers does not break up, the sending peer continuously pushes chunks of one (or multiple) sub-stream(s) to the receiving peer without further signaling messages.
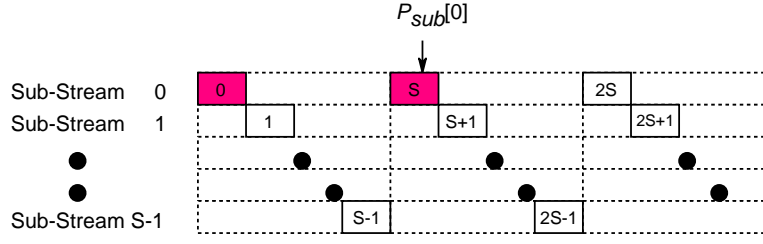
Fig. 1. Video segementation into sub-streams

In LStreaming, we combine two video streaming modes: tree-push and mesh-pull. In the tree-push mode, video sub-streams are pushed from the source to each peer. The delivery path of each sub-stream forms a delivery tree. In the mesh-pull mode, video chunks are retrieved by a peer from its neighbor individually. If a chunk is downloaded in the tree-push mode, we call it a pushed-chunk. Similarly, if a chunk is downloaded in the mesh-pull mode, we call it a pulled-chunk.

C. Buffer Management

To enable the tree-push mode and mesh-pull mode seamlessly at the same peer, the buffer of a LStreaming peer is organized as shown in Fig. 2. The total buffer is divided into three parts: push window, tolerance window and pull window. In the push window, chunks are expected to be pushed from other neighbors by requesting one or multiple sub-streams from one neighbor. Those missing chunks are downloaded in the mesh-pull mode in the pull window. In Fig. 2, the local peer requests three sub-streams from three neighbors. The tolerance window is introduced to avoid duplicate download in the tree-push mode and the mesh-pull mode. We apply a threshold-based heuristic to evaluate the push performance of a neighbor peer. If a neighbor cannot push a sub-stream quickly enough, the local peer may shift the download of this sub-stream from this neighbor to another neighbor.

In Fig. 2, P_{play} and P_{head} are the playback pointer and the head pointer, respectively. P_{play} indicates the chunk being played back. P_{head} indicates the downloaded chunk in the buffer with the largest ID. Note that some chunks between P_{play} and P_{head} may be still missing. In the current LStreaming implementation, when the peer downloads the first $10 \times U$ (i.e. 300) successive chunks after P_{play}, P_{play} starts to advance at the playback rate.

Related to P_{head}, the head chunks of sub-streams are denoted by a vector P_{sub}. $P_{sub}[i]$ points to the downloaded chunk of sub-stream i with the largest ID. This vector is included in the buffer map and sent to neighbors in order to avoid loops when constructing sub-stream trees. The starting point for the push window is $L \times U$ chunks (i.e. 60 chunks) behind P_{head}. P_{push} is computed as $\max\{0, P_{head} - L \times U\}$.

Up to now, the tree-push and mesh-pull modes are stiffly combined together. We introduce the tolerance window to seamlessly separate the chunk download into two modes. There exist unavoidable delays due to the chunk delivery
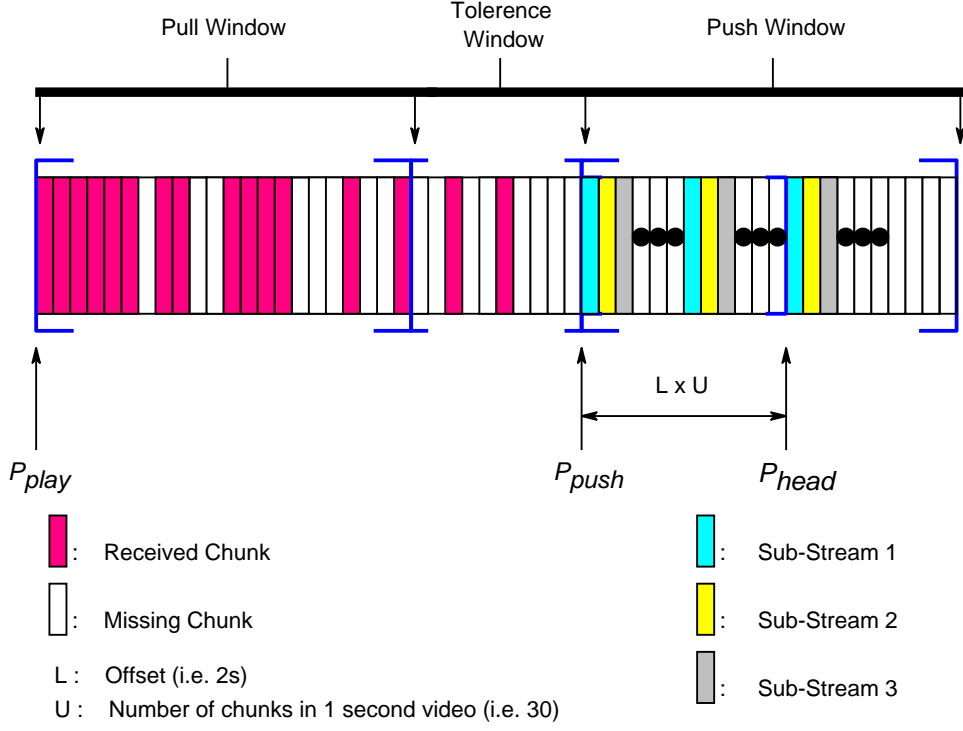
Fig. 2. Buffer Management of LStreaming

and the signaling process between senders and receivers. If there is no tolerance window between the push window and the pull window, a peer may download duplicate chunks from neighbors. For example, suppose one chunk is near the boundary of the push window and the pull window. One neighbor may have already been pushing it to a peer; however, the download has not been finished. Nevertheless, due to the advance of P_{head}, this chunk may fall into the pull window and this peer may initiate another pull request to download this chunk again. As a result, this chunk is downloaded twice. This toleration time interval is set to the average value of the RTT estimation between the local peer and its neighbors in LStreaming.

III. SYSTEM DESIGN AND IMPLEMENTATION

In this section, we present the system design and implementation details of LStreaming. In particular, we will illustrate the design considerations to reduce video redundancy. To facilitate the discussion, some notations are explained in Table I. The default values are used in our simulation and prototype if not specified explicitly.

A. Design consideration

The tree-push streaming is able to maintain high throughput and low overhead when peer churn is low. However, the tree structure is sensitive to network dynamics and does not work well in a dynamic environment. The throughput and overhead of a hybrid push-pull system highly depends on the tree-push component. A good hybrid system

TABLE I
NOTATION

R	playback rate (i.e. 300kbps)
v_i	upload rate of peer i (chunks / sec)
t_i	task token no. for neighbor peer i
P_{play}	playback pointer of the player
P_{head}	largest ID of the downloaded chunks in the buffer
P_{push}	starting ID of push window
$P_{sub}[i]$	head of sub-stream i
L	offset from P_{head} to P_{push} (i.e. 2s)
T_{re}	period of sub-stream re-scheduling (i.e. 10s)
T_s	initial buffering time (i.e. 20s)
U	number of chunks in 1 second video (i.e. 30)
S	total number of sub-streams (i.e. 15)

must have the ability to adapt to peer churn quickly, and further maintain the stable state. LStreaming achieves this adaptability via a closed-loop feedback control.

In Fig. 3, the feedback module in LStreaming consists of two mechanisms, the sub-stream scheduling and the re-scheduling. This module is not complicated but has the ability to catch up the network dynamics, schedule the sub-stream download to combat the network fluctuation, and eventually achieve the rapid convergence. The design philosophy behind the LStreaming is to adopt simple strategies to achieve this goal. Sharing the same design philosophy in [7], [8], we believe that a simple design is the most efficient in the running systems.
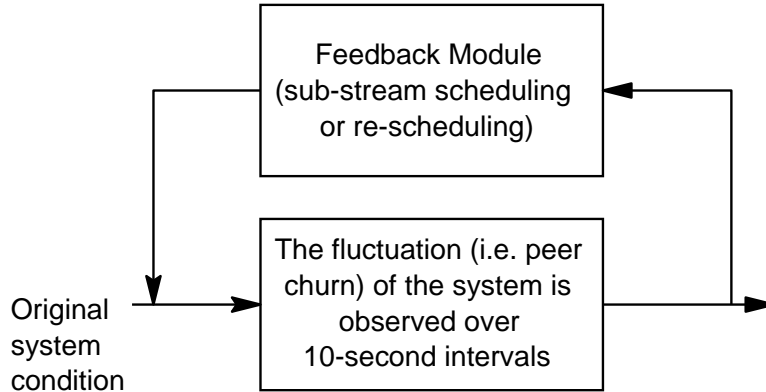

Fig. 3. The closed-loop feedback control in LStreaming

B. System initialization

When a peer first joins the channel, only the mesh-pull mode is enabled during the initial buffering time T_s. This peer exchanges buffer maps with other peers. Based on the harvested buffer maps, for missing chunks in the pull window, this peer sends the pull requests to download these chunks. After the initial buffering time (T_s), the

tree-push mode will be enabled as well. For the chunks in the push window, this peer requests sub-streams from its neighbors.

C. Sub-stream scheduling

In LStreaming, the original video stream is divided into a number of sub-streams with the same bit rate. The distribution of each sub-stream forms a delivery tree. A potential problem in the tree-push streaming is that peers may arbitrarily join a distribution tree without considering whether the upload capacity of a parent peer can support the streaming rate for its children. To avoid this problem, we propose a learning-based mechanism to assist peers to select an appropriate parent and join a sub-stream tree. This is the first step in the feedback module to adapt to network conditions.

After a peer issues a sub-stream request to a neighbor, this neighbor essentially pushes a series of chunks of this sub-stream without further notification. The goal of the sub-stream scheduling is to select an appropriate neighbor to push sub-streams. Suppose S sub-streams in the system. When a peer schedules sub-stream requests in the tree-push mode, it considers two issues: 1) how many sub-streams should one neighbor push? 2) where does this neighbor start to push chunks for one sub-stream? This whole process is called *sub-stream scheduling*.

During the initial buffering time, the peer has pulled some chunks from its neighbors. The download rate of these chunks from these neighbors can be used to estimate the upload rate for each neighbor. Suppose peer k has n neighbors and their estimated upload rates to peer k are denoted as $v_1, \ldots, v_n$. For each neighbor i, the maximum (token) number of sub-streams to be scheduled is calculated as follows:

$$t_i = \left\lceil \frac{v_i}{\sum_{i=1}^{n} v_i} \times S \right\rceil.$$

Because $\sum_{i=1}^{n} t_i$ may be larger than S, not all the neighbors is necessarily used to push sub-streams. Denote $h = \min\{h : \sum_{i=1}^{h} t_i \geq S\}$. It is sufficient to find the smallest h to cover S. For the first $h-1$ neighbors, request t_i sub-streams randomly from neighbor i. For the h-th neighbor, it is possible that the number of remaining sub-streams is less than t_h. In this case a peer requests all the remaining sub-streams from the h-th neighbor.

After a peer computes the number of sub-streams to request from each neighbor, it determines the beginning of each requested sub-stream. For sub-stream i, if $P_{sub}[i] \leq P_{push}$, the beginning chunk is the first missing chunk of sub-stream i, whose sequence ID is larger than or equal to P_{push}; otherwise, it is the next missing chunk of $P_{sub}[i]$ of sub-stream i. The "sub-stream scheduling messages", carrying the number of sub-streams and the IDs of the starting chunks, are sent to each corresponding neighbor.

D. Sub-stream Re-scheduling

Due to network dynamics in P2P networks, the upload rate from peer neighbors may fluctuate significantly. Though the sub-stream scheduling is achieved based on peers' previous achievable upload rates, a parent peer may not have a sufficient upload rate to deliver chunks to its children as time goes on. To achieve a good streaming performance in the push mode, one peer should periodically examine the achievable upload rates of its push-neighbors and re-schedule the sub-stream download from one neighbor to another neighbor if the original neighbor cannot sustain the upload rate of the assigned sub-streams. This mechanism mainly accomplishes the feedback function.

To evaluate the streaming performance of a push-neighbor, we introduce a simple threshold-based heuristic in LStreaming as follows. If the streaming rate for one sub-stream is below a threshold, the corresponding push-neighbor is canceled with the right to continue pushing chunks for this particular sub-stream after its receiving a canceling message. The local peer may request another neighbor to push this sub-stream. We call this operation as "*push-neighbor switching*".

1) Periodical Re-scheduling: In sub-stream re-scheduling, each LStreaming peer examines the number of pushed-chunks from each neighbor every T_{re} (i.e. 10s) and adjusts the workload for each neighbor based on their achievable upload rates in the previous T_{re}. For example, let $S = 15$, $U = 30$ and $T_{re} = 10$. Ideally, 20 chunks should be received from one sub-stream over T_{re}. If one peer receives more than 10 chunks of one sub-stream from one neighbor, we classify this neighbor as a qualified peer, who pushes chunks of this sub-stream to the local peer. This neighbor continues to push chunks of this sub-stream in the next T_{re}. Similarly, the local peer freezes the streaming of those sub-streams from the unqualified neighbors, who are not able to achieve the rate threshold. Then the local peer re-schedules these freezed sub-streams.

In this sub-stream re-scheduling, the local peer recalculates the maximum number (token) of sub-streams, which can be allocated to each neighbor based on the number of chunks downloaded from them in the previous $T_{re} = 10$ seconds. If one neighbor has already been qualified for pushing one sub-stream, then its token number is subtracted by 1. With the remaining token numbers, similar to the sub-stream scheduling, the local peer schedules the download of the freezed sub-streams from its neighbors.

2) Loop Avoidance: Due to the sub-stream re-scheduling, loops may occur in the system. To avoid loops for sub-stream i, when the local peer re-schedules sub-stream i, it only requests sub-stream i from the neighbor who has a larger head of sub-stream i than that of the local peer. This simple method can effectively avoid loops. However, the loop avoidance may lead to some sub-streams not allocated because no such neighbor can be found. Therefore, we introduce the *head detection* mechanism.

3) Head Detection: When the local peer receives a buffer map from one neighbor, it examines its un-scheduled sub-streams to see whether this neighbor has any sub-streams, whose head is larger than that of the local peer. We call this "head detection". If the answer is positive and this neighbor still has some token number left, this sub-stream may be requested to download from this neighbor. There exists one exception, though. If the current time is close to the next sub-stream re-scheduling round, the peer may wait until next sub-stream re-scheduling to schedule this sub-stream. In LStreaming, this tolerance time is set to 3 seconds empirically.

In case that some sub-streams can not be pushed due to loop avoidance, the video chunks of these sub-streams have to be downloaded in the pull mode.

E. Discussions

The push-pull hybrid streaming provides a new avenue in improving the performance of P2P streaming systems. Hybrid systems may be designed and implemented with different algorithms; however, the peer behaviors share similarities at large. Each time one peer retrieves a new chunk, as a push-neighbor, it pushes this chunk immediately to its children peers.

1) Push-neighbor switching: A push-neighbor continues to push chunks if it does not receive any sub-stream re-scheduling or canceling messages. Due to the sub-stream re-scheduling that changes push-neighbors, video chunks may be downloaded multiple times and become traffic redundancy. There are two types of duplicate chunks.

- Chunks in transmission: These chunks have been pushed out from the push-neighbors before the push-neighbor switching. Due to the propagation delay and the network buffering effect, they are still in transit. The amount of this redundancy only depends on the frequency of the push-neighbor switching.

- Chunks in the push-queue of push-neighbors: These chunks have been pushed out from the application layer of the push-neighbor, but still buffered in the packet queue of the operating system. Usually the sending of these buffered chunks cannot be withdrawn. Therefore, the number of this redundancy depends on both the frequency of the push-neighbor switching and the queue length.

2) Push-queue: In the push-pull streaming, in order to achieve high throughput and low overhead, chunks are preferred to be pushed in the tree-push mode. In general, chunks to be pushed and chunks to be pulled are stored in different application queues before transmission. Those chunks in the push-queue should be sent out with a higher priority by the operating system. If the length of this push-queue is set large, many chunks have the opportunities to be pushed out. However, when the push-neighbor switching occurs, chunks in this push-queue may still be delivered and cause redundancy since some chunks may be pushed from new push-neighbor again. If this push-queue is set small, less redundancy occurs when the push-neighbor switching happens. Nevertheless, more chunks

will be downloaded in the mesh-pull mode.

The length of this push-queue should be designed carefully. In our simulation and the prototype, we assume three types of DSL users with the upload capacities of 1Mbps, 512kbps and 128kbps. The 1Mbps peers can fully support 1Mbps/(300kbps/15)=50 sub-streams. When a peer receives one chunk, the probability of this chunk belonging to any sub-stream is 1/15, and this chunks will be pushed out to 3.3 ($50 \times (1/15) = 3.3$) children peers each time on average; therefore, the queue size is suggested to set to 4 in both the simulation and the prototype. This configuration reduces redundancy caused by chunks buffering in the push-queue when push-neighbor switching occurs.

With an appropriate length of the push-queue, the frequency of the push-neighbor switching is the dominating factor of the video redundancy. A small number of the push-neighbor switching is preferred in push-pull systems. In the next section, we will demonstrate that LStreaming is able to reduce the redundancy effectively.

IV. PERFORMANCE EVALUATION

In this section, We first evaluate the performance of LStreaming with a comparison to GridMedia and a generic random mesh-pull scheme via simulation. We developed a discrete-event simulator coded in C++ to capture the system behaviors at the chunk level. This simulator is implemented based on the simulator engine in [7]. We implement the LStreaming algorithm and re-use the implementation of GridMedia and the random mesh-pull algorithm for the performance comparison.

A. Simulation settings

In order to conduct a fair simulation comparison among different streaming algorithms, each peer runs the streaming algorithms under investigation simultaneously by introducing multiple "virtual peers" in the same peer. Each virtual peer implements one streaming algorithm. Those virtual peers using the same streaming algorithm form an independent P2P streaming network. These streaming networks in one simulation experiment are running in parallel without interfering with each other. In this design of simulation experiments, the simulation settings can be maintained exactly the same for evaluating different streaming algorithms, including network topology, link latency and peer churn.

To achieve a realistic latency setup in simulation, the end-to-end link latency between peers is randomly selected from the real-world node-to-node latency matrix (2500×2500) [9]. The mean end-to-end delay in this latency matrix is 79ms. The playback rate of the stream is 300kbps and the default neighbor number is 15. Similar to the simulation settings in [7], all peers are assumed to be DSL users with three types of the upload capacities of 1Mbps, 512kbps and 128kbps, and with the download capacities of 3Mbps, 1.5Mbps, 768kbps, respectively. The upload capacity of the video source is 900kbps. These three types of peers represent 10%, 50% and 40% of the total peers. We

simulate a flash-crowd event of 15 minutes. During the first 60 seconds, 6000 peers join the channel randomly. Then, we simulate two cases of peer behaviors in the simulation: static versus dynamic. In the static case, peers do not leave after they join the channel. In the dynamic scenario, 2000 peers leaves the channel during 400~500 seconds and another 2000 peers depart within 700~800 seconds randomly. The remaining 2000 peers stay in the channel until the end of the simulation. In this dynamic scenario, we aim to evaluate the robustness of the systems and examine the recovery performance on peer churn.

B. Simulation results

We provide a quantitative characterization of traffic breakdown in mesh-pull, GridMedia and LStreaming. We differentiate the total traffic overhead into two parts: signaling overhead and video redundancy overhead. After all the peers join the channel, we collect data statistics every 10 seconds on the streaming performance and the traffic overhead.

1) Download performance: We introduce two metrics to characterize the download performance using the following traffic ratios with respect to the video playback rate.

- *Total download ratio*: the ratio between the total download rate and the video playback rate.
- *Effective download ratio*: the ratio between non-duplicate video download rate and the video playback rate.

The *effective download ratio* characterizes the useful video download of the system. A smooth video playback is achieved when this ratio equals 1. The difference between the *total download ratio* and the *effective download ratio* is the *total traffic overhead ratio*. Since the overhead traffic consumes the network bandwidth, this overhead should be minimized. In Fig. 4, the average streaming performance over all the peers of mesh-pull, GridMedia and LStreaming is depicted. Note that the effective download ratios of both Gridmedia and LStreaming are higher than that of the random mesh-pull scheme. In addition, the effective download ratio of LStreaming is closer to 1 compared with GridMedia in both static and dynamic cases. LStreaming outperforms Gridmedia and Mesh-pull in terms of the download performance.

2) Overhead breakdown: The total overhead traffic consists of signaling messages and redundant video chunks. In the push-pull systems, the redundant video download is a potential problem. Video packets usually have much bigger sizes than control packets. If the number of duplicate video chunks is large, redundant video traffic becomes the major contributor in the total overhead traffic.

In Fig. 5(a), the total overhead ratios of both mesh-pull and GridMedia reach 7.8%, 9.0% in the static scenario and 8.4%,10.3% in the dynamic scenario, respectively. Nevertheless, the total overhead of LStreaming is only 6.0% and 6.5% in both scenarios. Compared with mesh-pull and GridMedia, the overhead reduction of LStreaming is 23.0%, 33.3% in the static case and 30.0%, 36.9% in the dynamic case, respectively.
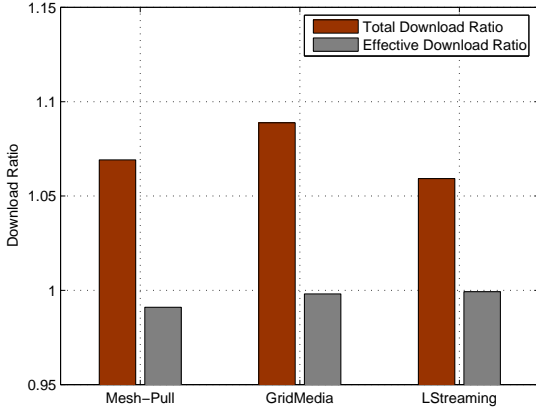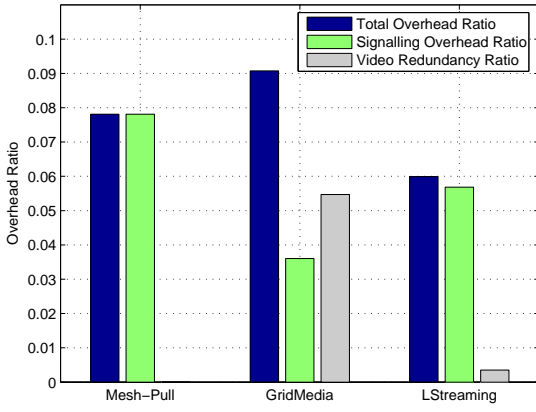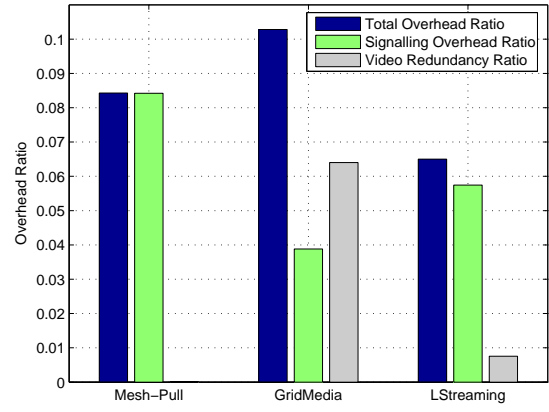
(a) Static
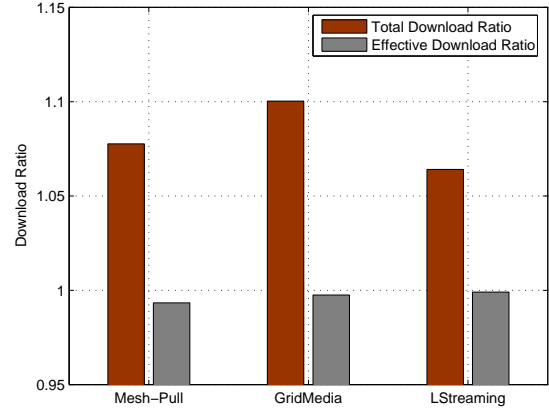
(b) Dynamic

Fig. 4. Performance of traffic download and total overhead

(a) Static

(b) Dynamic

Fig. 5. Overhead breakdown of download traffic

Due to the large video traffic volume, we believe that 10% overhead is significant and 1.3% increase from the static case to the dynamic case of GridMedia indicates the possible higher traffic overhead with more dramatic peer churn. The video redundancy is the major contributor in the total overhead in GridMedia as high as 5.4% out of the 9.0% total overhead download ratio in the static case, and 6.4% out of 10.3% in the dynamic case. Unlike GridMedia, LStreaming achieves a much smaller video redundancy ratio, 0.4% in the static case and 0.9% in the dynamic case, with a slightly higher signaling overhead. Because video packets are much larger than signaling packets, the total traffic overhead of LStreaming is lower accordingly, and its increasing rate from the static case to the dynamic case also maintains at a lower level.

3) Push-neighbor switching: This metric is defined as the average number of switching push-neighbors when peers conduct sub-stream re-scheduling. As we analyze previously, duplicate chunks are downloaded mainly due to the push-neighbor switching. In LStreaming, a low switching frequency leads to significant reduction of video redundancy. In Fig. 6, we plot the average number of the push-neighbor switching over each peer every 10 seconds in both cases. We observe that sub-stream scheduling and re-scheduling in LStreaming achieves load balancing very well. The system converges to a stable state faster and suffers less push-neighbor switching than GridMedia. The average values of LStreaming (see the lower dotted line in Fig. 6) in both cases are smaller than those of GridMedia (see the upper dotted line).
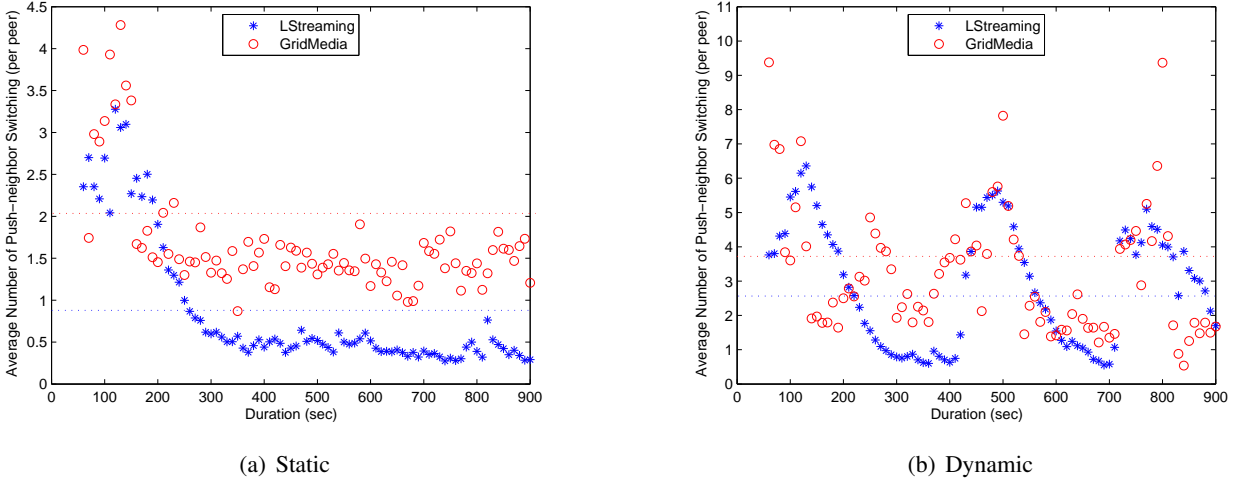

(a) Static   (b) Dynamic

Fig. 6.   The performance of the average push-neighbor switching

4) Quality ratio: We define the *quality ratio* as the ratio of the number of chunks, which has been played till the current time, and the number of chunks which should be played till the current time. This metric is computed every 10 seconds after all peers join the system ($t = 60s$). Fig. 7 shows the quality ratios of both GridMedia and LStreaming converge to 1 very quickly after the initial transition period. Nevertheless, the streaming performance of mesh-pull slightly suffers. In particular, in the dynamic case, when the peer churn is severe within 400~500 seconds and 700~800 seconds, LStreaming suffers the least and recovers the most quickly, compared with other two schemes.

C. Prototype experiment

We implement a prototype of LStreaming and conduct a preliminary evaluation of its performance on a campus network. In one experiment, 80 peers join a video channel of 300kbps. On the application level, we artificially limit the upload capacity of each client with the same distribution used in the simulation. The video source is capped
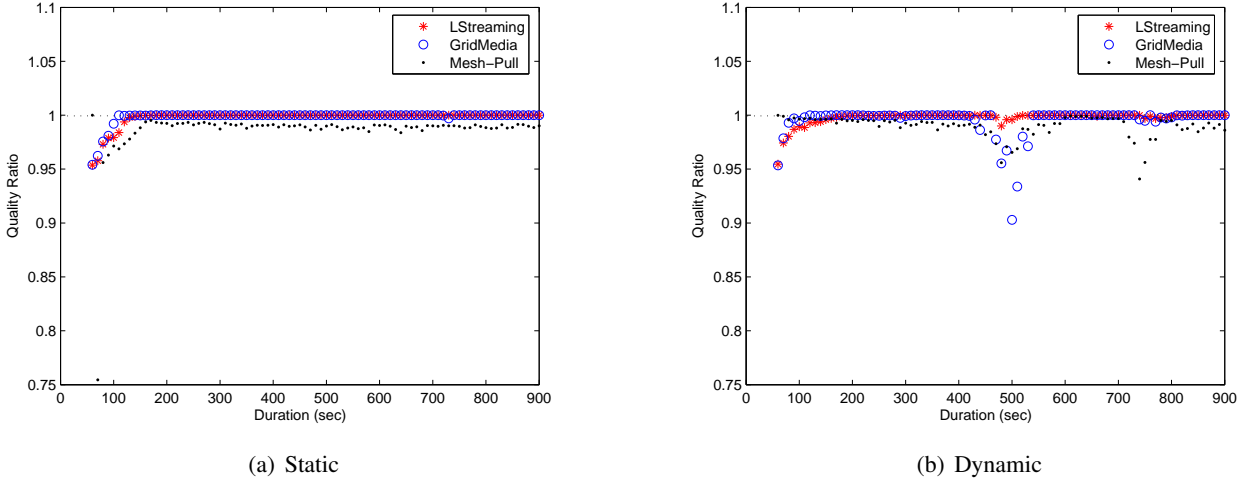
(a) Static   (b) Dynamic

Fig. 7. The performance of average streaming quality

with the upload capacity of 600kbps. The total streaming duration lasts for 50 minutes. All peers join the system within 5 minutes and 20 peers leave the channel at $t = 15$ minutes. As shown in Fig. 8, the average *quality ratio* is very close to the optimal value 1. The *quality ratio* fluctuates sightly when some peers leave and there is almost no visual impact during the experiment. We also illustrate the overhead breakdown for the prototype system in Fig. 8. Note that the video redundancy of the prototype matches well with the simulation results shown in Fig. 5. The total overhead ratio is as small as 4.5%, including only 0.8% video redundancy ratio. Fig. 8 demonstrates that LStreaming effectively reduces the total overhead, especially the video redundancy overhead; at the same time it achieves a very good streaming performance.
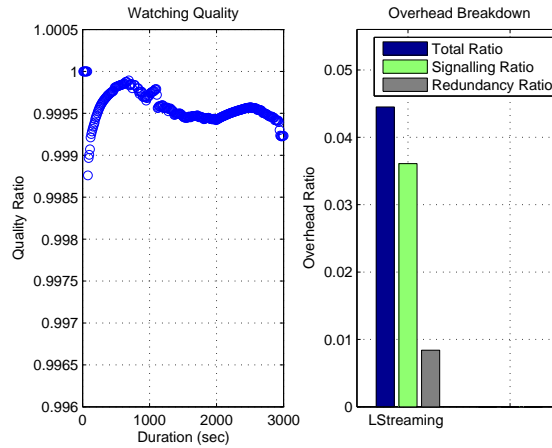

Fig. 8. The performance of the LStreaming prototype

The start-up time is another important performance metric in the live streaming. In this experiment, when one peer downloads 20-second continuous chunks, this peer starts to playback the video. In Fig. 9, we plot the cumulative

distribution function of the start-up time of the peers in this experiment. Note that 70% peers starts the video playback within 7 seconds. For a comparison, in a recent measurement study [10] on a popular P2P live streaming application, PPLive, the start-up delay is from 10 to 20 seconds; however, less popular channels had start-up delays of up to 2 minutes.
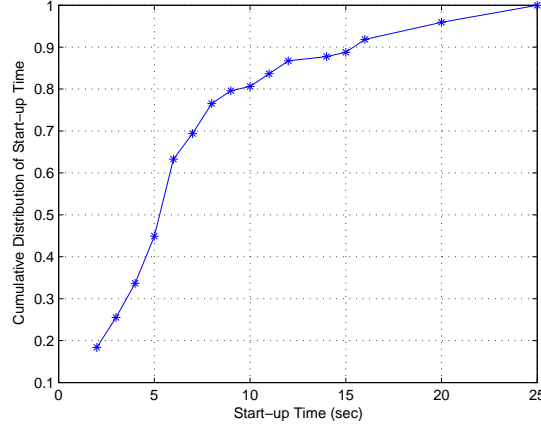

Fig. 9. The cumulative distribution function of the startup time

V. CONCLUSION

In this paper, we study the emerging push-pull P2P streaming approach. The push-pull streaming has the potential to achieve the optimal throughput and to reduce the signaling overhead; however, this push-pull streaming may incur significant traffic overhead if the system is not carefully designed. The video redundancy turns out to be the major contributor in the total traffic overhead. This redundancy is a direct consequence of the push-neighbor switching, in which the signaling asynchronization occurs between the peer and its push-neighbors, due to the propagation delay and the chunk buffering delay. This asynchronization cannot be completely avoided; nevertheless, with a low frequency of the push-neighbor switching, the video redundancy can be reduced significantly.

We design and implement simple but effective sub-stream scheduling and re-scheduling mechanisms in LStreaming in a closed-loop feedback system to achieve automatic load balancing with rapid convergence. The simulation results show that LStreaming has low frequency of push-neighbor switching. As a consequence, the total overhead is reduced by 23.0%, 33.3% in the static scenario and 30.0%, 36.9% in the dynamic scenario compared with generic mesh-pull and GridMedia. The prototype experiment demonstrates that LStreaming is practical and achieves the expected performance.

With the same push-neighbor switching frequency, the length of the push-queue may further impact the amount of the video redundancy. In our experiments, a suggested value is determined empirically in LStreaming. This

parameter also controls the proportion of video delivery in the tree-push mode and the mesh-pull mode. We are now investigating the tradeoff between mesh-pull and tree-push in the design of push-pull streaming systems. In addition, the current prototype experiments are only conducted on campus networks. We intend to continue wide-area experiments over the PlanetLab to evaluate the performance of LStreaming.

ACKNOWLEDGEMENT

REFERENCES

[1] Y.-H. Chu, S. G. Rao, S. Seshan, and H. Zhang, "A case for end-system multicast," *IEEE JSAC*, vol. 20, no. 8, pp. 1456–1471, Oct. 2002.

[2] X. Zhang, J. Liu, B. Li, and T.-S. P. Yum, "DONet/CoolStreaming: A Data-driven Overlay Network for Peer-to-Peer Live Media Streaming," in *IEEE INFOCOM*, vol. 3, Mar. 2005, pp. 2102 – 2111.

[3] J. Liu, S. G. Rao, B. Li, and H. Zhang, "Opportunities and challenges of peer-to-peer Internet video broadcast," *Proceedings of the IEEE*, vol. 96, no. 1, pp. 11–24, Jan. 2008.

[4] "PPLive," http://www.pplive.com.

[5] F. Pianese, D. Perino, J. Keller, and E. W. Biersack, "PULSE: An adaptive, incentive-based, unstructured P2P live streaming system," *IEEE Trans. on Multimedia*, vol. 9, no. 8, pp. 1645–1660, Dec. 2007.

[6] X. Hei, Y. Liu, and K. W. Ross, "IPTV over P2P streaming networks: the mesh-pull approach," *IEEE Communications Magazine*, vol. 46, no. 2, pp. 86–92, Feb. 2008.

[7] M. Zhang, Q. Zhang, L. Sun, and S. Yang, "Understanding the power of pull-based streaming protocol: Can we do better?" *IEEE JSAC*, vol. 25, no. 10, pp. 1640–1654, Dec. 2007.

[8] B. Li, S. Xie, G. Keung, J. Liu, I. Stoica, H. Zhang, and X. Zhang, "An empirical study of the coolstreaming+ system," *IEEE JSAC*, vol. 25, no. 10, pp. 1627–1639, Dec. 2007.

[9] "Merdian node to node latency matrix (2500×2500)," 2005, meridian project, http://www.cs.cornell.edu/People/egs/meridian/data.php.

[10] X. Hei, C. Liang, J. Liang, Y. Liu, and K. W. Ross, "A measurement study of a large-scale P2P IPTV system," *IEEE Trans. on Multimedia*, vol. 9, no. 8, pp. 1672–1687, Dec. 2007.